PATENT ABSTRACTS OF JAPAN

(11)Publication number : 07-225703
(43)Date of publication of application : 22.08.1995

(51)Int.Cl. G06F 11/28
G06F 9/06
G06F 11/30
G06F 12/14

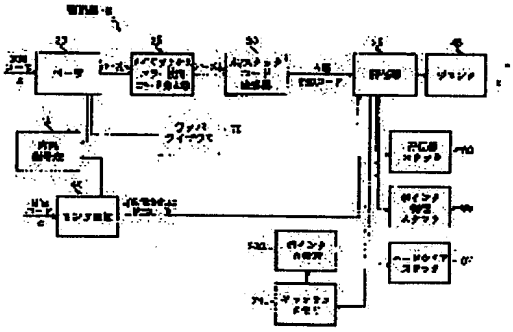
(21)Application number : 07-039426
(22)Date of filing : 06.02.1995

(71)Applicant : AT & T CORP
(72)Inventor : GOODNOW II JAMES E
KOWALSKI THADDEUS J
ROWLAND JAMES R

(30)Priority
Priority number : 94 192239 Priority date : 04.02.1994 Priority country : US

(54) DETECTION OF ERROR IN SOFTWARE PROGRAM

(57)Abstract:
PURPOSE: To permit a software program to detect an error for de-referring to an invalid pointer.
CONSTITUTION: Pointer information containing the content of a pointer which is known last and the valid memory range of the pointer is stored in a cache memory 74 on the respective pointers. Whenever the software program detects that a new value is substituted for the pointer, pointer information is updated. Whenever the software program de-refers to the pointer, an execution time pointer inspection sub-routine is mounted. The sub-routine retrieves pointer information on the de-referred pointer, obtains the actual content of the pointer from a memory position where the pointer is stored and judges whether a memory space indicated by the content is within the valid memory range or not unless the actual content is equal to the content recorded in pointer information.



LEGAL STATUS
[Date of request for examination]
[Date of sending the examiner's decision of rejection]
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number]
[Date of registration]
[Number of appeal against examiner's decision of

rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（A）**　(11)特許出願公開番号

特開平7－225703

(43)公開日　平成7年(1995)8月22日

(51)Int.Cl.[6]	識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｇ０６Ｆ　11/28	Ａ	7313－5Ｂ		
9/06	５４０　Ｕ	7629－5Ｂ		
11/30	３０５　Ｈ	7313－5Ｂ		
12/14	３１０　Ｈ			

審査請求　未請求　請求項の数28　ＦＤ　（全 44 頁）

(21)出願番号　特願平7－39426

(22)出願日　平成7年(1995)2月6日

(31)優先権主張番号　１９２２３９

(32)優先日　1994年2月4日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　390035493
エイ・ティ・アンド・ティ・コーポレーション
ＡＴ&Ｔ　ＣＯＲＰ.
アメリカ合衆国　10013－2412　ニューヨーク　ニューヨーク　アヴェニュー　オブ　ジ　アメリカズ　32

(72)発明者　ジェームズ　イー．グッドナウ　セカンド
アメリカ合衆国、95945　カリフォルニア、グラス　バレー、オールド　マイン　ロード　12477

(74)代理人　弁理士　三俣　弘文

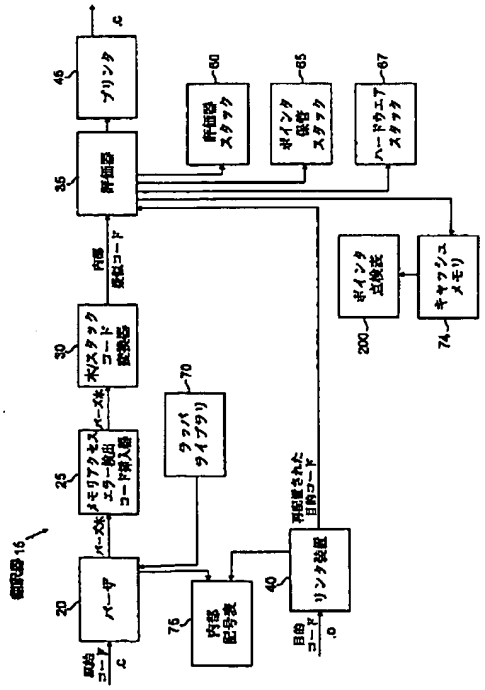
最終頁に続く

(54)【発明の名称】　ソフトウェアプログラムのエラー検出方法

(57)【要約】

【目的】　ソフトウェアプログラムが、無効なポインタをデレファレンスするエラーを検出する。

【構成】　各ポインタについて、最後に既知となったポインタの内容と、ポインタの有効メモリ範囲とを含むポインタ情報を格納しておく。ソフトウェアプログラムがポインタに新しい値が代入することを検出するごとにポインタ情報を更新する。次に、ソフトウェアプログラムによってポインタのデレファレンスが行われることを検出するごとに実行時ポインタ検査サブルーチンを実装する。このサブルーチンは、デレファレンスされたポインタに関するポインタ情報を検索し、そのポインタが格納されているメモリ位置からそのポインタの実際の内容を取得し、それがポインタ情報に記録されている内容と等しい場合に限り、その内容によって指示されるメモリ空間が、有効メモリ範囲内にあるかどうかを判定する。



1

【特許請求の範囲】

【請求項１】　テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される１個以上のライブラリ関数からなり、メモリ位置を指示する１個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、無効なポインタをデレファレンスするというエラーを検出する方法において、

前記ポインタのそれぞれについて、最後に既知となったポインタの内容と、ポインタの有効メモリ範囲とを含むポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの１個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、

前記ソフトウェアプログラムによってポインタのデレファレンスが行われることを前記ソフトウェア生成システムが検出するたびごとに実行時ポインタ検査サブルーチンを実装するステップとからなる方法であって、この実行時ポインタ検査サブルーチンが、

デレファレンスされたポインタに関するポインタ情報を検索する検索ステップと、

そのデレファレンスされたポインタが格納されているメモリ位置からそのポインタの実際の内容を取得するステップと、

その実際の内容が、前記ポインタ情報に記録されている当該デレファレンスされたポインタの最後に既知となった内容と等しい場合に限り、その実際の内容によって指示されているメモリ空間が、ポインタ情報から取得される有効メモリ範囲内にあるかどうかを判定するテストを行うテストステップとからなることを特徴とする、ソフトウェアプログラムのエラー検出方法。

【請求項２】　ポインタを作成することが既知であるライブラリ関数の実行後に、そのライブラリ関数によって作成されるポインタに対するポインタ情報を格納するステップを実行する実行後ラッパ関数を実行するステップをさらに有することを特徴とする請求項１の方法。

【請求項３】　前記ソースコード内のポインタに代入するためのポインタ値を返値として返すことが既知であるライブラリ関数の実行後に、前記ソースコード内の当該ポインタへのその返値の代入を検出したときのポインタ情報を更新するステップの前に、前記ソフトウェア生成システムが後で使うためにその返されたポインタに関する更新されたポインタ情報をポインタ保存スタックに格納するステップを実行する実行後ラッパ関数を実行するステップをさらに有することを特徴とする請求項１の方法。

【請求項４】　ポインタをデレファレンスすることが既知であるライブラリ関数を実行する前に、前記実行時ポインタ検査サブルーチンを開始する実行前ラッパ関数を

2

実行するステップをさらに有することを特徴とする請求項１の方法。

【請求項５】　制限された範囲を有する１個以上の引数を受け取るライブラリ関数を実行する前に、その受け取った引数がその制限された範囲内にあるかどうかをテストする実行前ラッパ関数を実行するステップをさらに有することを特徴とする請求項１の方法。

【請求項６】　前記ポインタ情報がポインタのステータス表示をさらに含み、

前記テストステップで、前記デレファレンスされたポインタの実際の内容が前記デレファレンスされたポインタの最後に既知となった内容と等しくない場合に前記ソフトウェア生成システムによって検出されなかったライブラリ関数の実行中の代入によって前記デレファレンスされたポインタの値が変更されたことを示すように、前記デレファレンスされたポインタに関するポインタ情報内のステータス表示を設定するステップをさらに有することを特徴とする請求項１の方法。

【請求項７】　前記ソフトウェアプログラムの実行後に前記ソフトウェア生成システムによって検出されなかったライブラリ関数の実行中の代入によって前記デレファレンスされたポインタの値が変更されたことを示しているステータスを有するポインタを識別するステップと、

ライブラリ関数によるポインタの変更を検出する実行後ラッパ関数が必要であることをプログラマに示すステップとをさらに有することを特徴とする請求項６の方法。

【請求項８】　前記ポインタ情報がポインタのアドレスをさらに含み、

前記検索ステップが、前記デレファレンスされたポインタに関するポインタ情報を識別するためにそのポインタ情報に含まれるポインタアドレスを検索するステップをさらに有することを特徴とする請求項１の方法。

【請求項９】　前記方法が、前記ソースコードに対応する構文木に対して読み出し時エラー検査プロセスを実行するステップをさらに有し、その読み出し時エラー検査プロセスが、

前記ソースコード内に現れるポインタ代入ごとに、ポインタテーブル更新コマンドと、ポインタに代入されるポインタ式を特徴づける特徴情報とを、前記構文木に挿入する読み出し時ポインタテーブル更新サブルーチンを実行するステップと、

前記ソースコードに現れるポインタのデレファレンスごとに、前記構文木にポインタ検査コマンドを挿入する読み出し時ポインタ検査サブルーチンを実行するステップとからなることを特徴とする請求項１の方法。

【請求項１０】　前記構文木に対してデータフロー解析を実行するステップをさらに有することを特徴とする請求項９の方法。

【請求項１１】　前記データフロー解析の結果を利用して、前記構文木から、重複するポインタ検査コマンドを

除去することを特徴とする請求項１０の方法。

【請求項１２】　前記データフロー解析の結果を利用して、不正なポインタのデレファレンスを実行時に検出することを特徴とする請求項１０の方法。

【請求項１３】　前記データフロー解析の結果を利用して、未初期化ポインタのデレファレンスを実行時に検出することを特徴とする請求項１０の方法。

【請求項１４】　前記構文木を内部擬似コードに変換するステップをさらに有することを特徴とする請求項９の方法。

【請求項１５】　前記方法が、前記内部擬似コードに対して実行時エラー検査プロセスを実行するステップをさらに有し、この実行時エラー検査プロセスが、

読み出し時にポインタ代入ごとに前記構文木に挿入されたポインタテーブル更新コマンドおよび特徴情報を評価したときに、前記ポインタ情報を格納するステップおよび更新するステップを実行する、実行時ポインタテーブル更新サブルーチンを実行するステップと、

読み出し時にポインタのデレファレンスごとに前記構文木に挿入されたポインタ検査コマンドを評価したときに、前記実行時ポインタ検査サブルーチンを実行するステップとからなることを特徴とする請求項１４の方法。

【請求項１６】　テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される１個以上のライブラリ関数からなり、メモリ位置を指示する１個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、割当てメモリ領域を解放した後にその領域を指示するポインタをデレファレンスすることによりその領域にアクセスするというエラーを検出する方法において、

前記割当てメモリ領域を指示するポインタのチェインリストを含む、前記割当てメモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの１個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、

前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、

前記割当てメモリ領域の解放を前記ソフトウェア生成システムが検出した場合に、対応する割当て情報からチェインリストを取得するステップと、

当該チェインリスト内にリストされた各ポインタに関するポインタ情報内に、そのポインタは解放したメモリ空間を指示しているという表示を記録するステップと、

ポインタがデレファレンスされるごとに対応するポインタ情報を評価して、そのポインタが解放されたメモリ空間を指示しているという表示があるのにもかかわらずデレファレンスされているのかどうかを判定するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項１７】　テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される１個以上のライブラリ関数からなり、メモリ位置を指示する１個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、メモリ領域の未初期化部分を指示するポインタをデレファレンスすることによりその未初期化部分を読み出すというエラーを検出する方法において、

メモリの各バイトに対する初期化ステータスを含む、前記メモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記メモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの１個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、

前記ソースコードファイルがポインタをデレファレンスすることによってメモリのバイトにアクセスしてそのバイトを初期化することを前記ソフトウェア生成システムが検出するたびごとに前記初期化ステータスを更新するステップと、

ライブラリ関数によって前記メモリ領域の１個以上のバイトが初期化されることを検出して、そのような初期化が検出された場合に前記初期化ステータスを更新する初期化ビットベクタ保守サブルーチンを実行するステップと、

ポインタがデレファレンスされて前記メモリ領域の１個以上のバイトが読み出されるたびごとに前記割当て情報内の初期化ステータスを評価して、読み出されるすべてのバイトが初期化されているかどうかを判定するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項１８】　テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される１個以上のライブラリ関数からなり、割当てメモリ領域を指示する可能性のある１個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、割当てメモリをリークするというエラーを検出する方法において、

前記割当てメモリ領域を指示するポインタのチェインリストを含む、前記割当てメモリ領域に関する割当て情報

5

を格納するステップと、
前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、
前記ソフトウェアプログラムによって前記ポインタのうちの１個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、
前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、
チェインリストが空である場合にメモリリークであると識別するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項１９】　ソフトウェア生成システムによって実行されるソフトウェアプログラムによって呼び出される関数に渡されるポインタであって前記呼び出される関数内の形式引数に代入される実引数に関連し前記ソフトウェアプログラムおよび前記呼び出される関数内のポインタごとに前記ソフトウェア生成システムによってポインタ検査テーブルに保持されるポインタ情報を前記ソフトウェアプログラムから前記呼び出される関数に渡す方法において、
前記実引数に関連する渡されるポインタ情報の一時的格納用にメモリ位置を割り当てるステップと、
前記ポインタ検査テーブルから、前記実引数に関連するポインタ情報を取得するステップと、
取得したポインタ情報を、前記ソフトウェアプログラムによって前記呼び出される関数が呼び出されたときに前記一時的格納用メモリ位置に格納するステップと、
前記呼び出される関数の開始とともに、前記一時的格納用メモリ位置から前記ポインタ情報を取得するステップと、
前記呼び出される関数内の形式ポインタ引数に関連するポインタ検査テーブル内のエントリに、取得したポインタ情報を挿入するステップとからなることを特徴とする、ソフトウェアプログラムから呼び出される関数にポインタ情報を渡す方法。

【請求項２０】　ソフトウェア生成システムによって実行されるソフトウェアプログラムによって呼び出された関数からそのソフトウェアプログラムに返されるポインタ値に関連し前記ソフトウェアプログラムおよび前記呼び出された関数内のポインタごとに前記ソフトウェア生成システムによってポインタ検査テーブルに保持されるポインタ情報を返す方法において、
前記呼び出された関数によって返されるポインタ情報に関連するポインタ情報の一時的格納用にメモリ位置を割り当てるステップと、
前記ポインタ検査テーブルから、前記呼び出された関数

6

から返されるポインタ値に関連するポインタ情報を取得するステップと、
取得したポインタ情報を、前記呼び出された関数から前記ソフトウェアプログラムに復帰したときに前記一時的格納用メモリ位置に格納するステップと、
前記呼び出された関数への復帰後に、前記一時的格納用メモリ位置から前記ポインタ情報を取得するステップと、
前記ソフトウェアプログラムの後続の実行中に、前記返されたポインタ値が代入されたポインタに関連するポインタ検査テーブル内のエントリに、取得したポインタ情報を挿入するステップとからなることを特徴とする、呼び出された関数からソフトウェアプログラムにポインタ情報を返す方法。

【請求項２１】　メモリ位置を指示する１個以上のポインタを含むソフトウェアプログラム内の各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるソフトウェアプログラム内の、それぞれポインタ式を有する複数の命令ストリームを含みそれらの命令ストリームのうちの１つのみが実行時にソフトウェア生成システムによって実行されその結果が実行時にデレファレンスされる複合式に対して、ポインタ検査を実行する方法において、
ポインタ情報の一時的格納用メモリ位置を割り当てるステップと、
前記ソフトウェア生成システムが、実行される命令ストリーム内のポインタに関するポインタ情報を取得し、取得したポインタ情報を前記一時的格納用メモリ位置に挿入するというコマンドを各命令ストリームに挿入するステップと、
実行時に前記複合式のポインタがデレファレンスされたときに前記一時的格納用メモリ位置からポインタ情報を取得するステップとからなることを特徴とするポインタ検査方法。

【請求項２２】　メモリ位置を指示する１個以上のポインタを含み、各ポインタの有効メモリ範囲の情報を含む各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるライブラリ関数が、無効なポインタをデレファレンスするというエラーを検出する方法において、
前記ライブラリ関数によってデレファレンスされる各ポインタに関するポインタ情報を取得し、デレファレンスされるポインタの実際の内容によって指示されるメモリ空間がポインタ情報から取得した有効メモリ範囲内にあるかどうかを判定するラッパ関数を前記ライブラリ関数と関係づけるステップと、
前記ライブラリ関数を実行する前に前記ラッパ関数を実行するステップとからなることを特徴とする、ソフトウェアプログラムのエラー検出方法。

【請求項２３】　メモリ位置を指示する１個以上のポイ

7

ンタを含みソフトウェア生成システムによって実行されるライブラリ関数の実行中に作成されるポインタに関するポインタ情報を更新する方法において、
前記ライブラリ関数によって作成されるポインタに関するポインタ情報を格納するラッパ関数を前記ライブラリ関数と関係づけるステップと、
前記ライブラリ関数の実行後に前記ラッパ関数を実行するステップとからなることを特徴とする、ポインタ情報の更新方法。

【請求項２４】　ソフトウェア生成システムによって実行され、メモリ位置を指示する１個以上のポインタを含むライブラリ関数およびソースコードについて、ライブラリ関数からソースコードに返されソースコード内のポインタに代入されるポインタ値に関連しポインタごとにポインタ検査テーブルに保持されるポインタ情報を更新する方法において、
返されるポインタ値に関するポインタ情報をメモリ位置に格納するラッパ関数を前記ライブラリ関数と関係づけるステップと、
関係づけられたライブラリ関数の実行後に前記ラッパ関数を実行するステップと、
返されたポインタ値が前記ソースコード内のポインタに代入されたときに、前記メモリ位置からポインタ情報を取得して前記ポインタ検査テーブルに挿入するステップとからなることを特徴とするポインタ情報の更新方法。

【請求項２５】　テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される１個以上のライブラリ関数からなり、メモリ位置を指示する１個以上のポインタを含み、実行するソフトウェアプログラム内の各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるソフトウェアプログラムに対して実行されるポインタ検査の数を最小にするためにソフトウェアプログラムをデータフロー解析する方法において、
前記ソフトウェアプログラム内の各ポインタに対するフローセットを設定するステップと、
前記ソフトウェアプログラムの各行に対して解析ルーチンを実行するステップと、
ポインタのデレファレンスを有するフローセットごとに、拡張オフセットを有する１回のポインタ検査を実行するステップとからなり、前記解析ルーチンが、
解析しているソフトウェアプログラムの行内のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに新しいフローセットを設定するステップと、
デレファレンスされるポインタに対する現在のフローセットに、そのデレファレンスされるポインタが解析しているソフトウェアプログラムの前記行内でデレファレンスされたことの表示によりマークするステップと、
前記ソフトウェアプログラムの各行に対して前記解析ルーチンを繰り返すステップとからなることを特徴とする、ソフトウェアプログラムをデータフロー解析する方法。

8

【請求項２６】　メモリのバイトごとに初期化ステータスを保持するソフトウェア生成システムによって実行されるソフトウェアルーチンによるメモリのバイトの初期化を検出する方法において、
前記ソフトウェアルーチンがアクセスする可能性のあるすべての領域のバイトを識別するステップと、
前記領域の各バイトを既知の値でマークするステップと、
すべてのアクセス可能メモリに対する初期化ステータスを解析し、前記ソフトウェアルーチンが実行されるときに未初期化であるすべてのバイトを識別するステップと、
前記ソフトウェアルーチンが実行される前にすべての未初期化メモリの第１のチェックサムを実行するステップと、
前記ソフトウェアルーチンが実行された後にすべての未初期化メモリの第２のチェックサムを実行するステップと、
第１のチェックサムが第２のチェックサムに等しくない場合に前記ソフトウェアルーチンによるメモリの初期化があったと検出するステップとからなることを特徴とする、ソフトウェアルーチンによるメモリの初期化を検出する方法。

【請求項２７】　メモリのバイトごとに初期化ステータスを保持するソフトウェア生成システムによって実行されるソフトウェアルーチンによるメモリのバイトの初期化を検出する方法において、
前記ソフトウェアルーチンがアクセスする可能性のあるすべての領域のバイトを識別するステップと、
すべてのアクセス可能メモリに対する初期化ステータスを解析し、前記ソフトウェアルーチンが実行されるときに未初期化であるすべてのバイトを識別するステップと、
アクセス可能なすべての未初期化メモリを保存するステップと、
前記ソフトウェアルーチンが実行される前にすべての未初期化メモリの第１のチェックサムを実行するステップと、
前記ソフトウェアルーチンが実行された後にすべての未初期化メモリの第２のチェックサムを実行するステップと、
第１のチェックサムが第２のチェックサムに等しくない場合に前記ソフトウェアルーチンによるメモリの初期化があったと検出するステップとからなることを特徴とする、ソフトウェアルーチンによるメモリの初期化を検出する方法。

【請求項２８】　割当てメモリ領域を指示する１個以上

のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが割当てメモリ領域を解放する方法において、

前記割当てメモリ領域を指示する各ポインタのステータス表示およびチェインリストを含む、前記割当てメモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、ステータス表示および前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの１個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、

前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、

前記割当てメモリ領域の解放を前記ソフトウェア生成システムが検出した場合に、対応する割当て情報からチェインリストを取得するステップと、

当該チェインリスト内にリストされた各ポインタに関するポインタ情報のステータス表示内に、そのポインタは解放したメモリ空間を指示しているという表示を記録するステップとからなることを特徴とする、割当てメモリ領域を解放する方法。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】本発明は、ソフトウェアプログラムのテストおよびデバッグ処理システムに関し、詳しくは「翻訳された原始（ソース）コード」と「コンパイルされた目的（オブジェクト）コード」との両方におけるメモリアクセス誤り（エラー）を検出するための方法および装置に関する。

【０００２】

【従来の技術】今日のコンピュータプログラミングの世界においては、多くの高水準のプログラミング言語が利用できる。しかし、その中でＣ、Ｃ＋＋、およびＰａｓｃａｌのようなプログラミング言語は、それらの持つ柔軟性およびパワーによってプログラマの間で非常に人気のある言語となっている。

【０００３】これらのプログラム言語には、ソフトウェアにおいてプログラマが実現できることについての制約がもしあったとしても非常に少ないので、根底にあるアセンブリプログラムが行うタスクであれば実際上どのようなタスクでもプログラムに行わせることが可能である。

【０００４】顕著なパワーおよび柔軟性の得られるこれらのプログラミング言語の特徴の１つは、ポインタによって制約なしにメモリにアクセスできるという能力である。しかしポインタを制約なしに使用できることから、

従来のデバッグ処理手法では検出および訂正がしばしば困難であるようなプログラムバグが生じる。

【０００５】種々のメモリアクセスエラーの検出用に、いくつものソフトウェアテストおよびデバッグ処理ツールが開発されている。例えば、米国カリフォルニア州サニーベイルのピュアソフトウェア・インコーポレーテッドから商業的に入手可能で、米国特許第５，１９３，１８０号に開示されている、商品名「ピュリファイ(Purify)」ソフトウェアテストツールは、メモリアクセスエラーおよびメモリ漏洩（リーク）を検出するためのシステムを提供する。この「ピュリファイ」システムは、メモリの各バイトについてメモリ割付および初期化の状態を監視（モニタ）する。

【０００６】これに加えて、「ピュリファイ」システムは、メモリ要素配列（メモリ配列）境界違反および同様のメモリアクセスエラーを検出するために、割付を受けたメモリの各ブロックの前後に８バイトのバッファゾーンを設けている。バッファゾーン内の各バイトの状態は、割付も初期化もされていない状態（非割付・非初期化状態）にセットされる。

【０００７】メモリにアクセスする各命令について、「ピュリファイ」システムは、割り付けられていないメモリへの書き込み、および初期化されていないまたは割り付けられていないメモリからの読み出しをプログラムが行っていないことを確実にするための試験を行う。

【０００８】「ピュリファイ」システムを用いることによって、多くのメモリアクセスエラーを検出するための有効な基礎は得られるが、このシステムは、割付されたメモリの第１のブロックに連関するポインタが、割付され初期化されたメモリの第２のブロックに不適切にアクセスする場合に生じる共通プログラムエラーの検出はしない。

【０００９】すなわち、「ピュリファイ」システムは、ポインタによって指示されたメモリが割付されていることおよび初期化されていることだけを検証し、ポインタによって指示されたメモリがそのポインタについて設定されている適切な境界範囲内にあることの検証は行わない。

【００１０】この制限を克服しようとする試みから、別のソフトウェアテストおよびデバッグ処理ツールが開発された。例えば、ステッフェン(Joseph L. Steffen)の論文「ポータブルＣコンパイラへの実行時（ランタイム）検査の追加」（Software-Practice and Experience誌、第２２（４）巻、１９９２年４月、３０５－３１６ページ）に記述されている、コンパイラを基盤とするメモリアクセス検出システムは、各ポインタについて３語を利用し、それによってポインタがそのポインタの有効範囲についての情報を得られるようにしたものである。

【００１１】すなわち、ポインタがメモリにアクセスする都度、そのポインタによって指示されたメモリがそれ

ぞれのポインタについての適切な境界範囲内にあることを確実にするような検査が行われる。

【0012】しかし、プログラマの多くは、自分のソフトウェアのテストおよびデバッグ処理を、コンパイラ環境よりも翻訳器（インタプリタ）環境で行うことを好む。その理由は、インタプリタを基盤とするデバッグ処理処理の方が一般に、程度の高い追跡およびその他の診断手法が得られることから、デバッグ処理中により柔軟性がありまたより大きな支援が得られるためである。

【0013】たぶん、より普通なのは、いくつかのファイルが「翻訳された原始コード」から構成され他のファイルが「コンパイルされた目的コード」から構成されるような、部分的翻訳である。

【0014】しかし、部分的翻訳の環境で作動するソフトウェアデバッグ処理ツール（例えば、センタラインソフトウェア・インコーポレーテッドから商業的に入手可能な、前に「セーバ・C」システムとして知られた「センタラインコードセンタ」システム）は一般に、「コンパイルされた目的コード」のエラー検査に関しては、上に述べた「ピュリファイ」システムと同じ制限がある。

【0015】詳しくは、これらの部分的翻訳デバッグ処理ツールは一般に、「コンパイルされた目的コード」内でポインタによって指示されたメモリが割付され且つ初期化されていることだけを検証し、ポインタによって指示されたメモリがそのポインタについて設定されている適切な境界範囲内にあることの検証は行わない。

【0016】

【発明が解決しようとする課題】従来の技術についての上記の欠点から明らかなように、「翻訳された原始コード」と「コンパイルされた目的コード」との両方を実行しながらエラー検出タスクを行うことのできるソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。さらに、与えられたポインタによって指示されたメモリがそれぞれのポインタについての適切な境界範囲内にあることを確実にするソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。

【0017】その上、読み出し時またはパーズ時に導出される情報を利用することにより、実行時に行われる重複または重合するポインタ検査の数を減少させるような、より効率的なソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。

【0018】

【課題を解決するための手段】概略的には、本発明の一態様によれば、ソフトウェアプログラムにおけるいくつものメモリアクセスエラー、を検出するためのソフトウェアテストおよびデバッグ処理システムとしての処理ツールが設けられる。これらのメモリアクセスエラーは例えば、配列次元違反、無効ポインタの「関連付けの解除（関連解除）（デレファレンス）」、解放されたメモリへのアクセス、初期化されていないメモリからの読み出し、およびメモリ漏洩の自動検出である。これに加えて、本システムは、検出されたエラーの特定および訂正を行うための診断情報を有する。

【0019】本発明の別の態様によれば、ソフトウェアプログラムにおける各ポインタについてのポインタ情報を記録するためのシステムが設けられる。記録されたポインタ情報は、例えば連関するポインタのアドレスおよび内容、ならびに連関するポインタが有効に指示できるメモリ範囲を含むのが好ましい。

【0020】これに加えて、割付されたメモリの各領域についての割付情報を記録するためのシステムが設けられる。記録されたメモリ割付情報は、例えば割付されたメモリの連関領域を現に指示する全ポインタのチェインリスト、および割付されたメモリの連関領域の各バイトの初期化状態を維持する初期化ビットベクトルを含むのが好ましい。

【0021】読み出し時エラー検査プロセスが、翻訳された原始コードに連関する構文木（パーズ木）の各ラインを解析する。翻訳された原始コードに現れる各ポインタ割当またはポインタ関連解除の各々に応答して、「読み出し時エラー検査プロセス」が、パーズ木内にエラー検査コマンドおよび追加情報を適切に挿入する。

【0022】「実行時エラー検査プロセス」が、ポインタ割当に応答してパーズ木内に挿入された各エラー検査コマンドについての適切な、記録されたポインタ情報の更新を開始する。同様に、ポインタ関連解除に応答してパース木内に挿入された各エラー検査コマンドについて、「実行時エラー検査プロセス」が、関連解除（デレファレンス）されたポインタについて記録されているポインタ情報のポインタ検査を開始する。

【0023】実行時ポインタ検査は、関連解除されたポインタについての記録されているポインタ情報を検討評価（または単に、評価）し、関連解除されたポインタがその有効メモリスペース外を指示しているかどうかを検出する。

【0024】本発明のさらに別の特徴によれば、「コンパイルされた目的コード関数」（または簡単に、「コンパイルされたコード関数」）に対するエラー検査プロセスが、エラー検査を要する「コンパイルされたコード関数」に連関する翻訳された「ラッパ関数」を用いて行われる。必要なエラー検査プロセスを実現するために、ラッパ関数は、必要に応じて、連関する「コンパイルされたコード関数」の前および／または後に実行される。

【0025】関連解除されたポインタの各々についてポインタ検査を開始するために、「コンパイルされた目的コード」の実行中ポインタを関連解除するものとして知られる「コンパイルされたコード関数」の各々に「実行前ラッパ関数」を連関させるのが好ましい。「実行前ラッパ関数」は、「コンパイルされたコード関数」へ送られた引数（アーギュメント）についての追加検査も行

う。

【0026】「コンパイルされたコード」の実行中ポインタの生成を行うものとして知られる「コンパイルされたコード関数」の各々に「実行後ラッパ関数」を連関させるのが好ましく、同様に、ポインタ値を返すものとして知られる「コンパイルされた目的コード関数」の各々に「実行後ラッパ関数」を連関させるのが好ましい。このポインタ値は次に、呼び出し関数によって返されるとすぐに別のポインタに割り当てられる。

【0027】もし連関させた「コンパイルされたコード関数」がポインタを生成する場合、「実行後ラッパ関数」が、生成されたポインタについてのポインタ情報を適切なメモリ位置に記録することが好ましい。もし連関させた「コンパイルされたコード関数」がポインタ値を呼び出し関数に返す場合、呼び出し関数の実行中に次に行われる検索取り出しのために、「実行後ラッパ関数」が、必要なポインタ情報をポインタ保存スタックに位置させることが好ましい。

【0028】本発明のさらに別の特徴によれば、割付を解除されたメモリの領域がアクセスされる場合を検出するための方法が提供される。割付されたメモリの領域の割付の解除後すぐに、そのメモリ領域に連関するメモリ割付情報に含まれるチェインリストがアクセスされて、割付されたメモリの領域を現に指示する全ポインタのリストが得られる。その後、ポインタが、割付を解除されたメモリの領域を今指示している旨の表示が、チェインリスト中に列記された各ポインタに連関するポインタ情報に記録される。

【0029】したがって、もし、割付を解除されたメモリスペースをポインタが指示している旨の表示がありながら、そのポインタが関連解除される場合に、エラーが検出される。

【0030】本発明の別の特徴によれば、初期化されていないメモリの領域からの読み出しが行われる場合を検出する方法が提供される。割付されたメモリの領域の1個以上のバイトが「翻訳された原始コード」によって初期化される都度、メモリのその連関領域についてのメモリ割付情報における初期化ビットベクトルが、その対応する数のメモリバイトが初期化されたことを表示するように更新される。

【0031】メモリを初期化する「コンパイルされたコード関数」が実行される都度、初期化ビットベクトル維持サブルーチンが行われる。初期化ビットベクトル維持サブルーチンは、「コンパイルされたコード関数」によってメモリの1バイト以上の初期化を検出し、その対応する数のメモリバイトが初期化されたことを表示するように初期化ビットベクトルを更新する。

【0032】メモリの領域からの読み出しを行うようにポインタが関連解除される都度、初期化ビットベクトルが評価されて、読み出しが行われているメモリ領域が初期化された状態にあるかどうかが定められる。

【0033】本発明のさらに別の特徴によれば、メモリ漏洩が自動的に検出可能となる。割付されたメモリの各領域を現に指示しているポインタのリストがポインタの割当によって修正される都度、割付されたメモリのその領域に連関するメモリ割付情報に含まれるチェインリストが更新される。もしチェインリストが空きの場合、メモリ漏洩が識別される。

【0034】本発明のなお別の特徴によれば、データ流れ解析を利用して、読み出し時パーズ木についての重複すなわち重合するポインタ検査を削除することによって、実行時において行う必要のあるポインタ検査の数を最小化する処理が行われる。これに加えて、データ流れ解析により、空のまたは初期化されていないポインタの関連解除を読み出し時において検出することが可能となる。

【0035】

【実施例】本発明に基づくメモリアクセスエラー検出システムを図1に示す。ここに開示するメモリアクセスエラー検出システムは、ソフトウェア生成システムを基盤とするソフトウェアテストおよびデバッグ処理ツールである。この場合のソフトウェア生成システムは、翻訳器（インタプリタ）、またはコンパイラ、あるいはその他、原始コードを実行可能なフォーマットに変換する能力を有する類似のシステムとして実施される。

【0036】本メモリアクセスエラー検出システムは、種々のメモリアクセスエラーを検出するために、検査対象ソフトウェアプログラムを解析する。これらのメモリアクセスエラーは例えば、配列次元違反、無効ポインタの関連解除、解放されたメモリへのアクセス、初期化されていないメモリからの読み出し、およびメモリ漏洩の自動検出である。加えて、本システムは、検出されたエラーの特定および訂正を行うための診断情報を有する。

【0037】本発明を以下5つの主要項目に分けて説明する。第1に、導入部において、まず図1の概略ブロック図を参照して、本発明のメモリアクセスエラー検出システムについて述べる。またこの導入部では、概略的には図1にまた詳しくは図2に示すようなポインタ検査表200についての詳細説明も行う。図2に示すポインタ検査表200は、図3～図6を参照して説明するような連関事例におけるデータからなる。

【0038】第2に、図1に示すメモリアクセスエラー検出コード挿入器25によって実行されるような、本発明の「読み出し時エラー検査プロセス」を、［読み出し時エラー検査動作］と題する項目において、図7～図11を参照して説明する。本項目においてはその終りに、図23および図24を参照して、実行時において行う必要のあるポインタ検査の数を最小化することとある読み出し時エラーを検出できるようにすることとを目的として読み出し時において行われるデータ流れ解析、につい

て述べる。

【００３９】第３に、「コンパイルされた目的コード関数」（または簡単に、「コンパイルされたコード関数」）についてのエラー検査を実現するための好ましい実施例を、［コンパイルされた目的コードのエラー検査］と題する項目において説明する。本項目においては、連関する「コンパイルされたコード関数」についてのエラー検査を行うために必要に応じて実行される「翻訳されたラッパ関数」に関して述べる。コンパイルされたメモリ割付解除関数の各々に連関する好ましい「実行前ラッパ関数」についても図１２を参照して説明する。

【００４０】第４に、図１に示す評価器３５によって実行されるような、本発明の「実行時エラー検査プロセス」を、［実行時エラー検査動作］と題する項目において、図１３～図２２を参照して説明する。

【００４１】最後に、メモリ漏洩を検査するための方法、および検出されていないポインタ修正のソースを「コンパイルされたコード関数」によって特定するための方法を、［メモリ漏洩の特定および検出されていないポインタ修正のソースの特定］と題する項目において説明する。

【００４２】図１に示すように、メモリアクセスエラー検出システムは、商業的にＡＴ＆Ｔ社から入手可能な、Ｃプログラミング言語用のＣＩＮ翻訳器をエラー検査関数が得られるように改造した翻訳器のような、翻訳器（インタプリタ）１５を用いるのが好ましい。下でさらに述べるように翻訳器１５は「翻訳された原始コード」および「コンパイルされた目的コード」の両方を実行しながらエラー検出タスクを行う。

【００４３】翻訳器１５によって受信される原始コードを、読み出し時においてパーザ（構文解析器）２０が解析し、原始コードを周知のパーザ木（構文木）に変換する。これに加えて、原始コードがプログラムメモリに読み込まれると、内部記号表７５がパーザ２０によって形成される。

【００４４】内部記号表７５には、原始コードに定義される記号ラベルの各々についてのエントリが既知の方法で含まれる。内部記号表７５の各エントリからその連関する記号ラベルが判る。またこれらの各エントリは、それぞれの記号ラベルに割付されたメモリ位置のアドレスを含む。

【００４５】メモリアクセスエラー検出コード挿入器２５が設けられ、パーザ２０によって生成されたパーズ木上で、図７～図１１を参照して下にさらに説明するように読み出し時エラー検査プロセスを行う。下にさらに述べるように、メモリアクセスエラー検出コード挿入器２５が、原始コードから導出されたパーズ木を解析し、適切な場合にはエラー検査コマンドおよび追加情報をパーズ木の中に挿入する。この追加情報は実行時において評価される。

【００４６】それから、メモリアクセスエラー検出コード挿入器２５によって生成された修正パーズ木（挿入されたエラー検査コマンドおよび追加情報から構成される）が、「木からスタックへコード変換する「木／スタックコード変換器」３０に入力される。木／スタックコード変換器３０は、機械語内に本来ある内部疑似コードを生成する。

【００４７】上に述べたように、翻訳器１５は「翻訳された原始コード」および「コンパイルされた目的コード」の両方を実行しながらエラー検出タスクを行うことが好ましい。そのため、コンパイルされた目的コードを再配置するリンク装置４０が設けられる。目的コードが読み出し時においてリンク装置４０によってプログラムメモリにロードされると、内部記号表７５が、目的コードにおいて定義された各記号ラベルについてのエントリを含むように、上に述べた方法で更新される。

【００４８】実行時において、評価器３５が、インタプリタスタック６０を利用して、木／スタックコード変換器３０によって生成された内部疑似コードと、翻訳された内部疑似コードから呼び出されたコンパイルされた目的コードとを、さらに下で述べる方法で実行する。評価器３５の出力をモニタするために、プリンタ４５が設けられる。このプリンタ４５は、原始コードフォーマットで評価器３５によって実行される内部疑似コードプログラムを列記する。

【００４９】本発明の一特徴によれば、評価器３５が、読み出し時中および実行時状況において内部疑似コード内に設定されたコマンドおよび情報に基づいて、図２を参照してさらに下で述べるように、ポインタ検査表２００を生成する。ポインタ検査表２００は、それぞれのポインタの使用および修正をモニタするために利用される各ポインタについての情報を記録する。

【００５０】これに加えて、ポインタ情報を評価器３５が将来使用することが予想されるのでそのために、ポインタ情報が、キャッシュメモリ７４、すなわち非常に高速のメモリ区域に評価器３５によって一時的に記憶される。本発明の一実施例においては、キャッシュメモリ７４が最も最近に参照された２個のポインタのアドレスを記憶する。これらの値が、下に述べるように実行時における動作中に評価器３５によって次に必要とされる場合、このような方法で、これらの値が評価器３５へ直接に転送されるので、動作速度が増加する。

【００５１】もし望むポインタ情報がキャッシュメモリ７４内に見出されない場合、好ましくは、バケットが最も最近のポインタ更新によって分類されるような「バケット・ハッシュ・ルックアップ」アルゴリズムを用いて、ポインタ検査表２００がアクセスされる。

【００５２】例えば、評価器３５によって実行されている関数の引数を記憶するために、周知のハードウェアスタック６７が評価器３５によってアクセスされる。これ

に加えて、評価器３５が、実行時エラー検査動作の好ましい実施例を実現化しながら、ポインタ保存スタック６５にアクセスする。これについてはさらに下に述べる。

【００５３】さらに下の「コンパイルされた目的コードのエラー検査」と題する項目で述べるが、コンパイルされたコード関数に連関する「翻訳された実行前および実行後ラッパ関数」を記憶するために、ラッパ・ライブラリ７０を設けることが好ましい。「翻訳された実行前および実行後ラッパ関数」によって、連関するコンパイルされたコード関数について、必要に応じてエラー検査プロセスを行うことが可能となる。

【００５４】さらに下で「実行時エラー検査動作」と題する項目において述べるように、ポインタ検査表２００の適切なエントリが、ポインタが新しい値を割り当てられる都度、評価器３５により実行時において更新される。この方法で、ポインタ検査表２００が、各ポインタについての現ポインタ情報を記録する。

【００５５】しかし、周知のように、従来の翻訳器は通常、コンパイルされた目的コードによってポインタ値に加えられた修正については気付かない。したがって、本発明の一態様により、ポインタ検査表２００内の適切なエントリを修正ポインタ情報で更新するために、ポインタ値を修正するものとして知られる連関するコンパイルされたコード関数の実行後に、下の「コンパイルされた目的コードのエラー検査」と題する項目において説明する「実行後ラッパ関数」を実行することが好ましい。

【００５６】これに加えて、新しいポインタ値が次に用いられるときにそのポインタ値についての前に検出されなかった修正を検出してフラグを付けるためのステップ１１２４およびステップ１１２８（図１８）における追補メカニズムが、下に図１８および図１９を参照して説明する実行時ポインタ検査サブルーチンによって得られる。

【００５７】図２に示すポインタ検査表２００のようなポインタ検査表は、異なるポインタに各々が連関する横列２２０、２２２、２２４のような、複数の横列を有する。ポインタ検査表２００の各横列は、連関するポインタについての情報を記憶するための複数のエントリを有する。

【００５８】エントリ２３０は、連関するポインタが記憶されているメモリ内のアドレスを記憶する。同様に、エントリ２４０は、連関するポインタの内容を記憶する。エントリ２５０ａは、連関するポインタの有効下側メモリ境界を記録し、他方、エントリ２５０ｂは、有効上側メモリ境界を記録を記録する。状態エントリ２６０は、連関するポインタについての、さらに下に述べるいくつもの予め定義されたコードのうちの１つを記録する。

【００５９】最後に、ファイル／ライン番号エントリ２７０は、ポインタの最後の修正に連関するファイル名およびライン番号を指示するポインタを記録する。この最後の修正データを利用してエラー検出時に診断情報が得られる。

【００６０】説明の便宜上、図３に示す原始コードファイル例「テストファイル」のライン１０からライン２１０までの実行後すぐに宣言され初期化されたポインタに連関するポインタ情報から構成されるものとする。なお、原始コードファイル「テストファイル」は、各整数変数について２バイトおよび各文字変数について１バイトを割付するマシン上で作動する。

【００６１】図３に示す原始コードファイル「テストファイル」のライン１０からライン４０までの実行中、整数番号「ｎｕｍｂｅｒ」および文字番号「ｎａｍｅ」を有する「ｐａｒｔ」型の構造体「ｗｉｄｇｅｔ」が宣言される。ポインタ「ｐｔｒ＿ｐａｒｔ」がライン５０において宣言される。

【００６２】ライン６０の実行中、ポインタ「ｐｔｒ＿ｐａｒｔ」は、構造体「ｗｉｄｇｅｔ」のアドレスを指示するように割当を受ける。ライン６０におけるこのポインタ指示割当によって、ポインタ「ｐｔｒ＿ｐａｒｔ」についてのポインタ情報がポインタ検査表２００内に入れられる。ライン１０からライン６０までの実行の結果として得られるメモリ割付を図４に示す。

【００６３】図４から判るように、ポインタ「ｐｔｒ＿ｐａｒｔ」にはアドレス「２０４８」がメモリ割付されており、ポインタ「ｐｔｒ＿ｐａｒｔ」についてのポインタ検査表２００のポインタエントリ２３０のポインタアドレス欄（図２）に入れられる。ポインタ「ｐｔｒ＿ｐａｒｔ」は、構造体「ｗｉｄｇｅｔ」のアドレス、すなわちアドレス「１０００」を指示するように割り当てられており、ポインタ「ｐｔｒ＿ｐａｒｔ」についてのポインタエントリ２４０のポインタ内容欄（図２）に入れられる。

【００６４】ポインタ「ｐｔｒ＿ｐａｒｔ」は、構造体「ｗｉｄｇｅｔ」内のどの位置、すなわち範囲１０００から１０１１までの領域内のどのアドレスを指示する割当も有効であるので、この情報は有効メモリ境界エントリ２５０ａ、２５０ｂ内に入れられる。

【００６５】ポインタ検査表２００内の状態エントリ２６０は、ポインタ「ｐｔｒ＿ｐａｒｔ」が「境界あり」（ＢＯＵＮＤＥＤ）とマーク付けした領域を指示することを表すように更新されることが好ましく、下でさらに述べるような方法で更新されることが好ましい。ファイル／ライン番号エントリ２７０は、ポインタ「ｐｔｒ＿ｐａｒｔ」がライン６０において原始コードファイル「テストファイル」によって前回更新されたことを表すように更新されることが好ましい。

【００６６】図３に示す原始コードファイル「テストファイル」のライン７０およびライン８０の実行中、２個の変数、すなわち整数変数「ｔｅｓｔＩｎｔ」および文

字変数「ｔｅｓｔｃｈａｒ」がそれぞれ宣言される。ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」がライン９０において宣言され、整数変数「ｔｅｓｔｉｎｔ」を指示するように割当を受ける。ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」がライン１００において宣言され、文字変数「ｔｅｓｔｃｈａｒ」を指示するように割当を受ける。

【００６７】ライン９０およびライン１００におけるポインタ割当によって、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」およびポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」についてのポインタ情報がポインタ検査表２００内に入れられる。ライン７０からライン１００までの実行の結果として得られるメモリ割付を図５に示す。

【００６８】図５から判るように、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」にはアドレス「２０８６」がメモリ割付されており、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」にはアドレス「２０９０」がメモリ割付されている。この情報は、各ポインタ、すなわちポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」およびポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」、についてのポインタ検査表２００のポインタエントリ２３０のポインタアドレス欄（図２）に入れられる。

【００６９】ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」は、整数変数「ｔｅｓｔｉｎｔ」のアドレス、すなわちアドレス「１５００」を指示するように割り当てられており、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」は、文字変数「ｔｅｓｔｃｈａｒ」のアドレス、すなわちアドレス「１５０２」を指示するように割り当てられている。この情報は、各ポインタ、すなわちポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」およびポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」、についてのポインタ検査表２００のポインタエントリ２４０のポインタ内容欄（図２）に入れられる。

【００７０】ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」は、２バイトの整数変数「ｔｅｓｔｉｎｔ」だけ、すなわち欄１５００から１５０１までの領域内だけを指示する指示するので、この情報は、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」についての有効メモリ境界エントリ２５０ａ、２５０ｂ内に入れられる。

【００７１】同様に、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」は、１バイトの文字変数「ｔｅｓｔｃｈａｒ」だけ、すなわち１バイトアドレス１５０２だけを指示する。したがって、この情報は、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」についてのポインタ検査表２００内の有効メモリ境界エントリ２５０ａ、２５０ｂ内に入れられる。

【００７２】ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」およびポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」についてのポインタ検査表２００内の状態エントリ２６０は、それぞれのポインタが「境界あり」の領域を指示することを表すように更新されることが好ましく、下でさらに述べるような方法で更新されることが好ましい。ファイル／ライン番号エントリ２７０は、これらのポインタがライン９０およびライン１００において原始コード「テストファイル」によって前回更新されたことを表すようにそれぞれ更新されることが好ましい。

【００７３】コンパイルされたライブラリ関数「ｍａｌｌｏｃ」を用いて３５バイトのメモリの割付を行い、割付を受けたメモリの始点アドレスの値を、宣言されたポインタ「ｐｔｒ＿ａｌｌｏｃ」に返す、原始コードファイル「テストファイル」のライン２００からライン２１０までの実行と、結果として得られるメモリ割付（図６に示す）とについて、下の「コンパイルされた目的コードのエラー検査」と題する項目において説明する。

【００７４】図３に示す原始コードファイル「テストファイル」のライン３００の実行中、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」（前に述べたようにライン１００において前に値を割り当てられている）が、構造体「ｗｉｄｇｅｔ」の構成部分（メンバ）を指示するように指示について再割当される。したがって、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」についてのポインタ検査表２００における情報については、指示割当に連関する新しいポインタ情報に更新する必要がある。

【００７５】すなわち、ライン３００の実行に続いて、ポインタ「ｐｔｒ＿ｔｅｓｔｃｈａｒ」についての「ポインタの内容」エントリ２４０を、ポインタが今、構造体構成部分（構造体メンバ）「ｗｉｄｇｅｔ．ｎａｍｅ」、つまり「１００２」を指示することを表すように更新する必要がある。

【００７６】なお、もしポインタ構成部分（ポインタメンバ）を含む第１の構造体が第２の構造体にコピーされる場合、第１の構造体におけるポインタメンバについてのポインタ検査表２００内に記録されているポインタ情報を、第２の構造体におけるポインタメンバについてのエントリ内に適切に入れる必要がある。

【００７７】本発明の一特徴によれば、ポインタ値の有効性を確実にするために、上に述べたポインタ検査表２００が、ポインタが関連解除される都度、すなわちそのポインタによって指示されるメモリがアクセスされる都度、利用される。例えば、図３に示す原始コードファイル「テストファイル」のライン３１０の実行後すぐに、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」によって指示されるアドレスに定数１２０２４が書き込まれるようにポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」が関連解除される。

【００７８】下でさらに述べるように、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」によって指示されるメモリ位置がアクセスされる直前に、ポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」が有効なメモリ位置、すなわちポインタ「ｐｔｒ＿ｔｅｓｔｉｎｔ」についての有効なメモリ境界エントリ２５０ａ、２５０ｂに記録されているような有効なメモリ境界１５００から１５０１までの範囲内、を指示していることを確実にするために、ポインタ内容の有効性

を検査する。メモリアクセスエラー検出システムが、

【００７９】本発明の好ましい一実施例において、静的ポインタについての情報を記録する表と自動ポインタについての情報を記録する表との２個の別個のポインタ検査表２００が維持される。自動ポインタは、自動ポインタが定義される特定関数の範囲内でだけ既知であるので、連関するポインタ情報を別個の表に記録する方が、より効率的である。したがって、自動ポインタ検査表２００は、各関数の実行後、再初期化される。

【００８０】ポインタ検査表２００には、２種類のポインタ、すなわち静的データを指示するように初期化される静的ポインタと主関数へ送られる「ａｒｇｖ」ポインタ引数とについてのポインタ情報をプリロードする（先に入れる）ことが好ましい。これらの原始コードファイルラインの実行は通常、下に述べるポインタ検査表更新のメカニズムに関連しないので、これらの情報については、内部疑似コードの実行に先立ってポインタ検査表２００内に自動的にプリロードすることが好ましい。

【００８１】［読み出し時エラー検査動作］メモリアクセスエラー検出コード挿入器２５がパーズ木の各ラインについて読み出しを行って、図７の流れ図に示す「読み出し時エラー検査プロセス」を起動実現させる。「読み出し時エラー検査プロセス」は、ある種のエラー検査コマンドと追加情報とをパーズ木に挿入するある種の読み出し時エラー検査タスクを行うために、パーズ木の各ライン、またはノード、を評価する。

【００８２】この、メモリアクセスエラー検出コード挿入器２５によって実現される「読み出し時エラー検査プロセス」は本来、図１３から図２２までを参照して下に説明するような、実行時において行われる実際のエラー検出プロセス、を起動し実行させる役をする前段階タスクを行う。

【００８３】「読み出し時エラー検査プロセス」は、図７に示すように、もし評価中のコードラインについて実行すべきエラー検査サブルーチンがあれば、そのうちのどのエラー検査サブルーチンを実行すべきかを定める。概略的には、プロセスについてのプログラム実行手順（以下簡単に、手順）は、パーズ木の各ラインが読み出されるステップ３００において「読み出し時エラー検査プロセス」に入り、その後、もしステップ３１０の条件が満足される場合、「読み出し時配列次元検査サブルーチン」（図８を参照して下に述べる）が実行される。

【００８４】同様に、もしステップ３２０の条件が満足される場合、「読み出し時ポインタ検査表更新サブルーチン」（図９および図１０を参照して下に述べる）が実行される。またもしステップ３３０の条件が満足される場合、「読み出し時ポインタ検査サブルーチン」（図１１を参照して下に述べる）が実行される。最後に、もしステップ３３５の条件が満足される場合、ステップ３３７の間に、適切なフラグ（下に述べる）がセットされる。

【００８５】これに加えて、メモリアクセスエラー検出コード挿入器２５がパーズ木のラインのすべてを読み出して解析したことがステップ３４０において検出されると、「読み出し時エラー検査プロセス」は、ステップ３５０およびステップ３６０においてデータ流れ解析（下でさらに説明する）を行う。これによって、実行時において行う必要のあるポインタ検査の数が削減され、読み出し時エラーの検出が可能となる。

【００８６】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、宣言された配列に対する参照を含むことがステップ３１０（図７）において検出された場合、手順はステップ４１０（図８）において図８に示す「読み出し時配列次元検査サブルーチン」に入り、同サブルーチンが開始される。

【００８７】「読み出し時配列次元検査サブルーチン」は、不正な添字（指標）、すなわち負の添字または宣言された配列サイズを超過する添字が配列を参照するために用いられているかどうかを判定する。多くのプログラミング言語において、多次元配列のうちの１つの次元の範囲を超過しながらなお配列の全範囲内に収まることは可能なので、各次元についてその適切な境界が維持されているかどうかの検査が行われる。

【００８８】なお、最大有効配列添字は一般に、その次元についての宣言された配列サイズよりも小さい。その理由は、多くのプログラミング言語について、配列の最初の有効添字が「０」であって「１」ではないからである。

【００８９】図８の「読み出し時配列次元検査サブルーチン（または簡単に、配列次元検査サブルーチン）」は、ステップ４１０において、内部記号表７５から配列の各次元についての、宣言されたサイズの最大値（最大宣言サイズ）を取得する。もし添字参照が可変の場合、添字参照の値は読み出し時においては判らず、実行時において参照が評価されるまでは評価できない。このため、配列のある次元についての添字参照が定数かまたは変数かを定める検査がステップ４２０においてまず行われる。

【００９０】もしステップ４２０において、添字参照が定数であると判断された場合、ステップ４３０において、コードのラインに現れる定数値がこの次元についての最大値と比較される。もしステップ４３０において、定数参照がこの次元についての最大宣言値を超える値かまたは負の値である場合、ステップ４３５においてエラーメッセージが生成され、ステップ４４０において、手順はプロセスから出て、プロセスが終了する。

【００９１】もしステップ４３０において、定数参照が有効であると判断された場合、手順はステップ４５０に進みここで、この配列に検査すべき他の次元があるかどうか、すなわちこの配列が多次元かどうかが判定され

る。

【００９２】もしステップ４５０において、検査すべき他の次元があると判断された場合、手順はステップ４２０に戻り、上に述べた方法で他の残りの次元についての検査が行われる。もしステップ４５０において、検査すべき他の次元が残っていないと判断された場合、手順はステップ３２０（図７）において「読み出し時エラー検査プロセス」に戻る。

【００９３】もしステップ４２０（図８）において、添字参照が可変であると判断された場合、ステップ４１０において内部記号表７５から検索により取り出されたそれぞれの次元についての許容最大サイズとともに、「ｄｉｍｃｈｋ」コマンドノードをステップ４２５においてパーズ木に挿入することが好ましい。「ｄｉｍｃｈｋ」コマンドノードは、次元参照が配列のこの次元についての有効な領域範囲内にあるかどうかを定めるための評価が実行時において行われる。

【００９４】「ｄｉｍｃｈｋ」コマンドノードがパーズ木に挿入されてしまうと、手順はステップ４５０に進みここで、この配列に検査すべき他の次元があるかどうかが上に述べた方法で判断され定められる。配列の次元がすべて検査されてしまうと、手順はステップ３２０（図７）において「読み出し時エラー検査プロセス」に戻る。

【００９５】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、ポインタに値を割り当てることがステップ３２０（図７）において検出された場合、手順はステップ５０５（図９）において図９および図１０に示す「読み出し時ポインタ検査表更新サブルーチン」に入り、同サブルーチンが開始される。

【００９６】ポインタ検査表２００がそれぞれのポインタについての新しい情報によって実行時において適切に更新されるように、ポインタに値が割り当てられる都度、「読み出し時ポインタ検査表更新サブルーチン」が、必要なコマンドおよび情報をパーズ木に挿入する。

【００９７】「読み出し時ポインタ検査表更新サブルーチン」が開始されると、ステップ５０５において、「ｔｂｌｕｐｄ」コマンドノードがパーズ木に挿入される。「ｔｂｌｕｐｄ」コマンドノードは、実行時において評価されると、ポインタ検査表２００の更新を開始する。実行時におけるポインタ検査表更新を行うために、読み出し時において取得できる追加情報、すなわち、ポインタに割り当てられるポインタ式の型の特性が、「ｔｂｌｕｐｄ」コマンドノードに入れられる。

【００９８】そして、ポインタに割り当てられるポインタ式が、ステップ５１０において解析され、ステップ５１５からステップ５６０（図１０）までにおいて、型のマッチングが得られるまで、いくつもの条件に対して検査される。

【００９９】ステップ５１５において、ポインタが、識別特定された変数または関数のアドレスを割り当てられているかどうかを定める検査が行われる。もしステップ５１５において、ポインタが、識別特定された変数または関数のアドレスを割り当てられていると判断された場合、ステップ５１８において、特定された変数または関数についての内部記号表７５内の適切なエントリを指示するポインタが、「ｔｂｌｕｐｄ」コマンドノードに入れられる。

【０１００】実行時において、内部記号表７５内のこのエントリが、必要なポインタ情報を得るためにアクセスされる。その後、手順はステップ３３０（図７）において「読み出し時エラー検査プロセス」に戻る。

【０１０１】もしステップ５１５において、ポインタが、識別特定された変数または関数のアドレスを割り当てられていないと判断された場合には、手順はステップ５２０に進む。

【０１０２】ステップ５２０において、ポインタが、第２のポインタの内容を割り当てられているかどうかを定める検査が行われる。もしステップ５２０において、ポインタが、第２のポインタの内容を割り当てられていると判断された場合、手順はステップ５２２に進む。この場合、実行時において第２のポインタが何を指示するか、および第２のポインタの有効な領域の範囲はどこかを読み出し時において判断することは不可能である。

【０１０３】しかし、実行時において、第２のポインタのアドレスはインタプリタスタック６０の最上部に位置する。したがって、ステップ５２２において、「ｒｅａｄｉｎｔｓｔｋ」コマンドが「ｔｂｌｕｐｄ」ノード内に入れられる。「ｒｅａｄｉｎｔｓｔｋ」コマンドは、実行時において評価されると、第２のポインタのアドレスを、さらに処理するために、インタプリタスタック６０から検索により取り出させる。その後、手順はステップ３３０（図７）において「読み出し時エラー検査プロセス」に戻る。

【０１０４】もしステップ５２０（図９）において、ポインタが、第２のポインタの内容を割り当てられていないと判断された場合、手順はステップ５２５に進む。

【０１０５】ステップ５２５において、ポインタが、記号列（ストリング）を割り当てられているかどうかを定める検査が行われる。もしステップ５２５において、ポインタが、ストリングを割り当てられていると判断された場合、割当を受けようとするポインタについてポインタ検査表２００の適切な横列に配置する必要のある情報、すなわちストリングのアドレスおよびサイズ、は読み出し時において既知である。

【０１０６】そして、ステップ５２７において、ストリングのアドレスおよびサイズが「ｔｂｌｕｐｄ」コマンドノード内に入れられる。その後、手順はステップ３３０（図７）において「読み出し時エラー検査プロセス」

に戻る。

【0107】もしステップ525（図9）において、ポインタが、ストリングを割り当てられていないと判断された場合、手順はステップ530に進む。

【0108】ステップ530において、ポインタが、不正値を割り当てられているかどうかを定める検査が行われる。もしステップ530において、ポインタが、不正値、すなわち定常的なゼロ、または負の値、を割り当てられていると判断された場合、ステップ532において、「不正」状態の表示が「tblupd」ノード内に入れられる。ポインタは不正値を記憶することが許容されるので、エラーメッセージはここでは生成されない。

【0109】しかし下でさらに述べるように、もしポインタが実行時において参照を除去され、しかもなお不正値を有する場合には、その時点においてエラーメッセージが生成される。ステップ532の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0110】もしステップ530（図9）において、ポインタが、不正値を割り当てられていないと判断された場合、手順はステップ535に進む。

【0111】ステップ535において、ポインタが、構造体メンバのアドレスを割り当てられているかどうかを定める検査が行われる。もしステップ535において、ポインタが、構造体メンバのアドレスを割り当てられていると判断された場合、構造体メンバのサイズは読み出し時において既知である。しかし、構造体メンバのアドレスは、実行時になるまで不明のままであり、実行時において構造体メンバのアドレスは、インタプリタスタック60の最上部に位置する。

【0112】そして、ステップ537において、構造体メンバのサイズおよび「readintstk」コマンドが「tblupd」ノード内に入れられる。「readintstk」コマンドが実行時において評価されると、構造体メンバのアドレスがインタプリタスタック60から検索により取り出される。ステップ537の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0113】もしステップ535（図9）において、ポインタが、構造体メンバのアドレスを割り当てられていないと判断された場合、手順はステップ540（図10）に進む。

【0114】ステップ540においてポインタが、複合式、例えばp＝＊（test？exp1：exp2）によって割り当てられているかどうかを定める検査が行われる。ここに、exp1およびexp2はポインタのアドレスを評価するポインタ式である。この特殊検査条件は、「if，then，else」ルーチン実現用のこの普通のプログラミング手法についてエラー検査を行うために実現される。

【0115】間接演算子「＊」は、検査条件が真かまたは偽りかに依ってexp1またはexp2のいずれかについて行われるので、これら2式のうちのどちらが新しいポインタの特性を定めるかについては、読み出し時においては未知である。

【0116】本発明の好ましい一実施例において、「savptr」コマンドのような命令が、2個の命令ストリームの各々に入れられる。実行時において実行されるのは、これら2個のうちの1個だけである。「savptr」コマンドは、実行された命令に連関するポインタについてのポインタ情報をポインタ保存スタック65に入れるように評価器35に命令する。このポインタ情報は、次に評価器35からアクセスされる。

【0117】ポインタ保存スタック65から検索により取り出されたポインタ情報は、ポインタ検査表200に入れるため、実行時において評価器35によって、下にさらに述べる方法で検索により取り出すことができる。

【0118】そして、もしステップ540において、ポインタが形式「＊（test？exp1：exp2）」の条件式によって割り当てられていると判断された場合には、ステップ542において、「savptr」コマンドが生成され、2個の式exp1およびexp2に連関する命令ストリームの各々に入れられる。

【0119】これに加えて、「rdptrstk」コマンドのような、第2の命令が「tblupd」ノードに入れられる。この第2の命令は、評価されると、評価器35をしてポインタ保存スタック65の内容を検索により取り出させる。ステップ542の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0120】もしステップ540（図10）において、ポインタが、このような複合式によって割り当てられていないと判断された場合には、手順はステップ545に進む。

【0121】ステップ545において、ポインタが、翻訳された関数への1個以上のポインタ引数の送付後すぐに暗黙のうちに割り当てられるかどうかを定める検査が行われる。プログラムが、翻訳された関数を呼び出すとき、「呼び出し」命令文に規定されるような、呼び出された関数の引数、すなわち実引数が、宣言された関数内のそれぞれの自動変数、すなわち形式引数に暗黙のうちに割り当てられる。

【0122】関数が呼び出されると、評価器35が、呼び出し関数についてメモリ割付を受けているハードウェアスタック67に、呼び出された関数の引数を押し込む。その後、呼び出された関数のための新しいスタックフレームが、評価器35により、スタックフレームポインタを更新することによって形成される。なお、呼び出された関数によって用いられるスタックフレームは、呼び出し関数のスタックフレームのアドレスとは異なるス

タックアドレスを持つことになる。

【0123】形式パラメータについてのポインタ検査表200内のエントリは、その関数へ送られた実ポインタパラメータ、すなわちポインタ、についてのポインタ検査表200内のエントリをコピーすることによって形成するのが好ましい。すなわち、ポインタである送られた実パラメータについてのポインタ検査表200からのポインタ情報がポインタ保存スタック65に入れられるのが好ましい。

【0124】このようにして、評価器35は、ポインタ保存スタック65内に入れられたポインタ情報をコピーすることによって、ポインタである型パラメータについて実行時においてポインタ検査表200内に適切なエントリを形成することができる。

【0125】そして、もしステップ545において、翻訳された関数へのポインタ引数の送付後すぐに暗黙のうちに割り当てられると判断される場合、手順はステップ547に進む。ここで、各ポインタ引数についての関数呼び出しに先立ち、「savptr」コマンドがパーズ木の中に入れられる。

【0126】本発明の好ましい実施例において、「savptr」コマンドは、送られたポインタ引数を受け取ることになる呼び出された「翻訳された関数」を指示するポインタを含む。「翻訳された関数」を指示するポインタは、ポインタ保存スタック65内に入れられた情報の有効性を確実なものにするための識別子の役をする。これに加えて、「rdptrstk」コマンドが「tblupd」ノードに挿入される。

【0127】「savptr」コマンドが実行時において次に評価されるとき、ポインタである「送られた実パラメータ」についてのポインタ情報がポインタ保存スタック65に入れられる。その後、実行時における「rdptrstk」コマンドの評価が、評価器35をして、ポインタ保存スタック65の内容を検索により取り出させる。この内容は、ポインタ検査表200に入れられる。ステップ547の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0128】もしステップ545において、翻訳された関数へのポインタ引数の送付後すぐに暗黙のうちに割り当てられることがないと判断される場合、手順はステップ550に進む。

【0129】ステップ550において、ポインタが、翻訳された関数から返された値を割り当てられるかどうかを定めるテストが行われる。読み出し時において関数から戻されることになる値についての情報は、もしあったとしても少ししかないので、返されたポインタについてのポインタ情報を実行時においてポインタ保存スタック65に押し込む命令がパーズ木に入れられる。

【0130】これによって、評価器35が、返されたポインタに連関するポインタ情報を、ポインタ検査表200に入れるために実行時においてポインタ保存スタック65から次に検索により取り出すことができる。

【0131】そして、もしステップ550において、ポインタが、翻訳された関数から返された値を割り当てられていると判断された場合、ステップ552において、呼び出された関数からの返呼に応答して、返されたポインタの各々について「savptr」コマンドが生成される。これに加えて、「rdptrstk」コマンドが、「tblupd」ノードに入れられる。

【0132】本発明の好ましい実施例において「savptr」コマンドは、ポインタ情報を返すポインタである「翻訳された関数」を指示するポインタを含む。この「翻訳された関数」を指示するポインタは、ポインタ保存スタック65内に入れられた情報の有効性を確実なものにするための識別子の役をする。その後、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0133】「savptr」コマンドが実行時において次に評価されるとき、関数から戻されるポインタに連関するポインタ情報がポインタ保存スタック65に入れられる。その後、実行時における「rdptrstk」コマンドの評価が、評価器35をして、ポインタ保存スタック65の内容を検索により取り出させる。この内容は、ポインタ検査表200に入れられる。

【0134】なお、関数がポインタを自動変数に返すことは不適切である。したがって、この場合にはエラーが発生する。

【0135】もしステップ550（図10）において、ポインタが、翻訳された関数から戻された値を割り当てられていないと判断された場合、手順はステップ560に進む。

【0136】もし手順がステップ560に到達した場合、ポインタに割り当てられているポインタ式は、上に設定されたテストのうちのどれか1つに基づく特徴付けができないことになる。すなわち、読み出し時においては、割り当てられるポインタについて何も確定せず、したがってステップ560において、「不明」状態の表示が「tblupd」ノードに入れられる。ステップ560の実行に続いて、手順は、ステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0137】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、少なくとも1回のポインタ関連解除を含むことがステップ330（図7）において検出された場合、手順は図11のステップ615において、図11に示す「読み出し時ポインタ検査サブルーチン」に入り、同サブルーチンが開始される。関連解除されたポインタの有効性をテストするために実行時においてポインタ検査表200にアクセスできるように、ポインタが関連解除される都度、「読み

出し時ポインタ検査サブルーチン」が、必要なポインタ検査コマンドをパーズ木に挿入する。

【０１３８】ステップ６１５において「読み出し時ポインタ検査サブルーチン」が開始されると、ステップ６１５において、関連解除されるポインタについて「ｐｔｒｃｈｋ」コマンドノードがパーズ木に挿入される。関連解除されるポインタの値は、ポインタの値が計算された後でメモリにアドレスするのにこの計算された値が用いられる前にはインタプリタスタック６０で最も上部にある項目なので、「ｐｔｒｃｈｋ」コマンドノードは、メモリアドレス命令の直前位置に入れられる。

【０１３９】「ｐｔｒｃｈｋ」コマンドノードが実行時において評価されると、図１８および図１９を参照して下で説明する「実行時ポインタ検査検査サブルーチン」が開始され、インタプリタスタック６０から検索により取り出された値が、それぞれのポインタについて、ポインタ検査表２００の有効なメモリ境界エントリ２５０ａ、２５０ｂに示されるようなそのポインタの有効領域範囲内にあるかどうかが判断される。

【０１４０】ステップ６２０において、評価中のパーズ木のラインに他のポインタ関連解除があるかどうかを定めるテストが行われる。もしステップ６２０において、評価中のパーズ木のラインに他のポインタ関連解除があると判断された場合、手順はステップ６１５に戻り、そこでさらに処理が続けられる。ステップ６２０において、評価中のパーズ木のラインにあるポインタ関連解除のすべてが処理されたと判断された場合、手順はステップ３３５（図７）において「読み出し時エラー検査プロセス」に戻る。

【０１４１】ステップ３３５においては、「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが関数への呼び出しを含むかどうかを判断するテストが行われる。もしステップ３３５において、評価中の原始コードファイルのラインが関数への呼び出しを含むと判断された場合、その関数によってアクセスされるメモリのすべてが、ステップ３３７において識別特定される。すなわち大域変数のすべておよびその関数に送られた変数のすべてである。

【０１４２】これに加えて、もしその関数がメモリにアクセスを有する場合には、メモリ割付アクセスフラグが、コンパイルされた関数に対応する内部記号表７５内のエントリにセットされる。図２１を参照して下に述べるように、このメモリ割付アクセスフラグをセットする動作が、連関する関数が実行時において呼び出される都度、「初期化ビットベクトル維持サブルーチン」を起動させる。

【０１４３】もしステップ３３５において、評価中の原始コードファイルのラインが関数への呼び出しを含まないと判断された場合、手順はステップ３４０に進む。

【０１４４】ステップ３４０においては、「呼び出し時エラー検査プロセス」が、まだ読み出され解析される必要のあるパーズ木の他のノードがあるかどうかを判断するテストを行う。もしステップ３４０において、まだ読み出され解析される必要のあるパーズ木の他のノードがあると判断された場合、手順はステップ３００に戻り、上に述べたように進む。しかし、もしステップ３４０において、まだ読み出され解析されるべきパーズ木の他のノードが残っていないと判断された場合、手順はステップ３５０に進む。

【０１４５】ステップ３５０においては、メモリアクセスエラー検出コード挿入器２５によって、修正パーズ木についてデータ流れ解析が行われる。修正パーズ木には、パーズ木に挿入されたエラー検査コマンドおよび追加情報を含む。データ流れ解析は、パーズ木についての重複すなわち重合するポインタ検査を削除することによって、実行時において行う必要のあるポインタ検査の数を最小化し、ある読み出し時エラーを検出できるようにする。

【０１４６】適切なデータ流れ解析アルゴリズムの記述については、Ｗ・ランディおよびＢ・ライダー（William Landi & Barbara G. Ryder）の論文「ポインタのある場合およびない場合のエーリアシング：問題分類学」（Center for Computer Aids for Industrial Productivity, Technical Report ＣＡＩＰ－１２５、ラトガーズ大学、１９９０年９月２５日）を参照されたい。

【０１４７】また同じく、Ｗ・ランディおよびＢ・ライダー（William Landi & Barbara G.Ryder）の論文「手続き間相互ポインタエーリアシングについての安全近似アルゴリズム」（SIGPLAN Notices、１９９２年７月、２３５～２４８ページ）を参照されたい。

【０１４８】図２３に示す原始コードファイル例「サンプル」のデータ流れ解析について下に簡単に述べる。データ流れ解析は、ポインタが新たな値を割り当てられまたは関連解除される都度、修正パーズ木を解析し識別特定する。なお、「ｔｂｌｕｐｄ」コマンドノードは前に、ポインタ割当の各々についてパーズ木に入れられており、「ｐｔｒｃｈｋ」コマンドノードは前に、ポインタが関連解除される都度パーズ木に挿入されている。

【０１４９】図２３に示す原始コードファイル「サンプル」は、構造体「ｗｉｄｇｅｔ」を指示するポインタ「ｐｔｒ＿ｐａｒｔ」を宣言し、初期化する。データ流れ解析は、評価中の原始コードファイルに現れる各ポインタについて流れセットを設定する。この流れセットは例えば、原始コードファイル「サンプル」の実行時に形成されたポインタ「ｐｔｒ＿ｐａｒｔ」について図２４に示すような、流れセットである。

【０１５０】与えられたポインタに対して、その連関するポインタが新たな値を割り当てられる都度、新たな流れセットが設定される。好ましい実施例においては、流れセットは、その連関するポインタが正規の値を割り当

てられる都度「ＤＥＦ」とマーク付け、すなわち定義され、その連関するポインタが不正値を割り当てられる都度「ＤＥＦ　ＩＬＬ」とマーク付けされる。同様に、その連関するポインタが関連解除される都度、「ＵＳＥＤ」（使用済）とマーク付けされる。

【０１５１】図２３に示す原始コードファイル「サンプル」のライン１０からライン５０までは、構造体「ｗｉｄｇｅｔ」を宣言する。ライン６０は構造体「ｗｉｄｇｅｔ」を指示するポインタ「ｐｔｒ＿ｐａｒｔ」を宣言し、ライン７０は構造体「ｗｉｄｇｅｔ」の始点を指示するためにこのポインタを初期化、すなわち定義する。ライン７０のデータ流れ解析によって、図２４に示すこのポインタ「ｐｔｒ＿ｐａｒｔ」についての第１の流れセットが「ＤＥＦ」とマーク付けされる。このマーク付けは、このポインタが「非ゼロ」値を割り当てられたことを表す。

【０１５２】構造体「ｗｉｄｇｅｔ」の３個のメンバに値を割り当てるために、ポインタ「ｐｔｒ＿ｐａｒｔ」がライン８０からライン１００までの各ラインにおいて関連解除される。そして、ライン８０からライン１００までの各ラインのデータ流れ解析によって、図２４における第１の流れセットが「ＵＳＥＤ」とマーク付けされる。このマーク付けは、このポインタが関連解除されたことを表す。

【０１５３】それからポインタ「ｐｔｒ＿ｐａｒｔ」がライン１５０において新たな値を割り当てられ、これによって、ポインタ「ｐｔｒ＿ｐａｒｔ」についての新たな流れセットが設定される。そして、ライン１５０ので流れ解析によって、図２４に示す第２の流れセットが「ＤＥＦ　ＩＬＬ」とマーク付けされる。このマーク付けは、このポインタが不正値を割り当てられたことを表す。

【０１５４】そして、構造体「ｗｉｄｇｅｔ」のメンバ「ｎｕｍｂｅｒ」に値を割り当てるために、ポインタ「ｐｔｒ＿ｐａｒｔ」がライン１６０において関連解除される。また、ライン１６０のデータ流れ解析によって、図２４における流れセットが「ＵＳＥＤ」とマーク付けされる。このマーク付けは、このポインタが関連解除されたことを表す。下で説明するように、この第２の流れセットの解析によって、読み出し時エラーが発生する。その理由は、ライン１６０が実行されるときに不正ポインタが関連解除中だからである。

【０１５５】パーズ木全体について流れセットが設定されると、実行時において行う必要のあるポインタ検査の数を最小化するために、各流れセットが解析される。定義によって、与えられたポインタは各流れセットの期間中同一の値を維持するので、各セットについて、各々の個別ポインタ検査を受け入れ可能なようにオフセットを拡張したポインタ検査を１回だけ行えばよい。

【０１５６】例えば、ポインタ「ｐｔｒ＿ｐａｒｔ」についての第１の流れセットには、このポインタの３回の関連解除が含まれる。すなわち、ポインタはライン７０において割り当てられた値を記憶している一方、関連解除は３回行われることになる。通常、データ流れ解析を行わない場合には、３回の別個のポインタ検査を行う必要がある。しかし、データ流れ解析によって、パーズ木における３回の別個のポインタ検査が、３回のポインタ関連解除の各々の有効性をテストする１回だけのポインタ検査に置換される。

【０１５７】本発明の好ましい実施例においては、上記データ流れ解析の正確さを確実なものにするための「コンパイルされたコード流れ解析ルーチン」が設けられ、このルーチンにおいては、与えられたポインタについての流れセットに、一連のポインタ関連解除とともに、コンパイルされた関数の呼び出しが含まれる。もしこのコンパイルされた関数が、この流れセットに連関するポインタへのアクセスを有する場合、このコンパイルされた関数は、翻訳器１５に気付かれずにこのポインタの値を修正できる。

【０１５８】そして、もしこのコンパイルされた関数がこのポインタへのアクセスを有する場合、実行時におけるコンパイルされた関数の実行後に、コンパイルされたコードの流れ解析ルーチンが、この流れセットに連関するポインタ値がコンパイルされた関数の実行中に修正されているかどうかを判断するテストを行うのが好ましい。

【０１５９】もしコンパイルされた関数がポインタ値を修正していた場合、このコンパイルされた関数の呼び出し後流れセットにおいて生じるこれらのポインタ関連解除についてポインタ検査を追加して再度実行する必要がある。もしコンパイルされた関数がポインタ値を修正していない場合、ポインタ検査を追加して行う必要はない。

【０１６０】実行時において行うべきポインタ検査の数を最小化することに加えて、上記のデータ流れ解析によって、あるエラーを読み出し時において検出することが可能となる。すなわち、ステップ３５０におけるデータ流れ解析の完了後、ステップ３６０において、初期化されていないまたは不正なポインタが関連解除されつつあることをこのデータ流れ解析が表示しているかどうかを判断するテストが行われる。

【０１６１】初期化されていないポインタを関連解除するというメモリエラーは、ポインタが割り当てられる前に関連解除されることを流れセットが表示する場合、すなわち流れセットが「ＤＥＦ」とマーク付けされる前に「ＵＳＥＤ」とマーク付けされる場合に生じる。同様に、不正なポインタを関連解除するというメモリエラーは、図２４の第２の流れセットにおけるように、流れセットが「ＤＥＦ　ＩＬＬ」とマーク付けされ、次に中間に来る割当なしに、「ＵＳＥＤ」とマーク付けされる場

合に生じる。

【0162】もしステップ360において、不正なまたは初期化されていないポインタが関連解除されていると判断された場合、ステップ370においてエラーメッセージが生成され、その後、手順はステップ380においてプロセスから出て、プロセスが終了する。

【0163】しかし、もしステップ360において、不正なまたは初期化されていないポインタが関連解除されてはいないと判断された場合、手順はステップ380において「読み出し時エラー検査プロセス」から出て、プロセスが終了する。

【0164】図7から図11までを参照して上に述べた「読み出し時エラー検査プロセス」および連関するサブルーチンが完了すると、パーズ木の各ラインの解析が終り、適切なエラー検査コマンドおよび情報のパーズ木への挿入が終る。そして、実行時エラー検査を起動実行するために、挿入されたエラー検査コマンドおよび追加情報が評価されることになる。

【0165】［コンパイルされた目的コードのエラー検査］本発明の別の特徴によれば、実行時において評価器35によって実行される、「コンパイルされた目的コード」についての同様なエラー検査を行うことも望ましい。

【0166】本発明の好ましい実施例において、「コンパイルされた目的コードのエラー検査」は、読み出し時において「翻訳された原始コード」について行われたエラー検査プロセスをまねるように、連関する「コンパイルされた関数」の実行時における実行の、必要に応じて前および／または後に、下に述べるように「翻訳されたラッパ関数」を実行することによって実現される。

【0167】好ましくは、すべての「コンパイルされたライブラリ関数」について、必要に応じてラッパ関数を設ける。これに加えて、ラッパ関数は、ポインタ値の監視および／または評価を要する、ユーザにより定義されるどの「コンパイルされた関数」、についても本明細書における教示に基づいて形成することができる。実行時における、連関する「コンパイルされた関数」の呼び出し時にラッパ関数が実行される実行方法について図20を参照して下に説明する。

【0168】評価器35によってアクセス可能なラッパライブラリ70を図1に示すように設けるのが好ましい。ラッパライブラリ70は、「コンパイルされたコード」の実行中にポインタを関連解除する「コンパイルされた関数」の各々について「実行前ラッパ関数」を維持することが好ましい。実行前ラッパ関数は、「コンパイルされたコード」の実行中に関連解除されるポインタのポインタ検査を、「翻訳された関数」においてポインタの関連解除のためにポインタ検査が行われるのと同じ方法で、起動する必要がある。

【0169】「実行前ラッパ関数」は、連関する「コンパイルされた関数」の実行の直前に実行するのが好ましい。これによって、ポインタが有効値を指示していなければそのポインタは関連解除されないということが確実になる。与えられた関数についての実行前ラッパ関数はまた、配列参照を含む送られた引数について、ここに述べるように配列次元検査をも行うことが好ましい。

【0170】同様に、もし与えられた「コンパイルされた関数」へ送られた引数が予め定義された有効範囲を有する場合、連関する「実行前ラッパ関数」は、この既知の範囲に対する送られた引数の有効性をテストすることができる。

【0171】なお、ある「コンパイルされた関数」については、連関する「実行前ラッパ関数」は、「コンパイルされた関数」の第1の実行時に「コンパイルされた関数」へ送られたポインタ引数またはその他の情報を記憶保存する必要がある。一方、同一関数への次の呼び出しは、前の実行時と同じ情報を用いる必要があることを表示するコードを送るだけである。

【0172】これに加えて、ラッパライブラリ70は、「コンパイルされたコード」の実行中にポインタを形成する「コンパイルされた関数」の各々について、および呼び出し関数に戻るときに次にポインタに割り当てられる値を返す「コンパイルされた関数」の各々について、「実行後ラッパ関数」を維持することが好ましい。「実行後ラッパ関数」は、「コンパイルされた関数」が戻る前に、しかし呼び出し関数が実行を再開する前に、実行することが好ましい。

【0173】もし「コンパイルされた関数」が、実行中にポインタを形成する場合には、「実行後ラッパ関数」は、その形成されたポインタについてのポインタ情報をポインタ検査表200に追加する必要があり、この追加は、ポインタ検査表エントリが「翻訳されたコード」によって形成されたポインタについて更新されるのと同一の方法で行う必要がある。

【0174】同様に、もし「コンパイルされた関数」が、次に呼び出し関数においてポインタに割り当てられることになる値を呼び出し関数に返す場合には、適切なポインタ情報をポインタ検査表200に追加する必要がある。この場合、返される値についてのポインタ情報をポインタ保存スタック65に、読み出し時ポインタ検査表更新サブルーチンがステップ552（図10）において「翻訳された関数」から返される値を取り扱ったのと同じ方法で入れる必要がある。

【0175】この後、ポインタ検査表200に入れる目的で、この情報がポインタ保存スタック65から評価器35によって検索により取り出される。

【0176】例えば、ポインタには、Cプログラミング言語の関数ライブラリに普通に見出される「ｍａｌｌｏｃ」関数のような、コンパイルされたメモリ割付関数から戻される値が頻繁に割り当てられる。「ｍａｌｌｏ

c」関数についての「実行後ラッパ関数」は、割付されたメモリ領域の始点アドレスと、始点アドレスおよびサイズ情報から導かれるこの領域の有効範囲とをポインタ保存スタック６５に入れることが好ましい。

【０１７７】このようなふうにして、返された値が呼び出し関数におけるポインタに割り当てられるとき、ポインタ検査表２００（図２）における適切な横列が、関連するポインタ情報データ更新される。始点アドレスがポインタエントリ２４０の内容欄に入れられ、有効範囲エントリ２５０には、「実行後ラッパ関数」によって計算された範囲情報が入れられる。好ましくは、「割付済」の状態表示を状態エントリ２６０に入れるようにする。

【０１７８】しかし、本発明の好ましい実施例によれば、割付されたメモリスペースを指示するポインタについて追加ポインタ情報が維持される。図３に示す原始コードファイル例「テストファイル」のライン２００およびライン２１０が実行されると、３５バイトのブロックが割付される結果となり、割付されたブロックの始点アドレスがポインタ「ｐｔｒ＿ａｌｌｏｃ」に割り当てられる。

【０１７９】「ｍａｌｌｏｃ」関数から戻ると、ポインタ「ｐｔｒ＿ａｌｌｏｃ」に対するポインタ検査表２００内の横列が、新たなポインタ情報を反映するように更新される。ポインタアドレスエントリ２３０とポインタ内容エントリ２４０には、適切な情報が入れられる。状態エントリ２６０には、割付されたメモリスペースをポインタが指示することを表す表示「割付済」（ＡＬＬＯＣＡＴＥＤ）が入れられる。

【０１８０】この好ましい実施例においては、「ｍａｌｌｏｃ」関数についての「実行後ラッパ関数」もまた、図６に示すようなメモリ割付構造体２８０を形成する。図２に示すように、ポインタ「ｐｔｒ＿ａｌｌｏｃ」についての有効下側メモリ境界２５０ａはメモリ割付構造体２８０を指示するポインタを有する。

【０１８１】メモリ割付構造体２８０は、下側境界部分２８２、上側境界部分２８４、チェインリスト部分２８６、初期化ビットベクトル２８８、および状態部分２９０からなる。メモリ割付の有効上側および下側境界は、構造体部分２８２および２８４にそれぞれ保存記憶される。

【０１８２】チェインリスト部分２８６は、割付されたメモリの連関ブロックを現に指示するすべてのポインタのリストからなる。割付されたメモリを指示するポインタが他のポインタにコピーされる都度、第２のポインタに対するポインタ検査表２００内の状態エントリ２６０も「割付済」とマーク付けされ、有効下側境界エントリ２５０ａ内の同じメモリ割付構造体２８０を指示するポインタを有する。これに加えて、新たなポインタがチェインリスト部分２８６内のリストに追加される。

【０１８３】同様に、もし割付されたメモリを指示するポインタが、割付されたメモリの新たなブロックを指示するように指示について再割当された場合、このポインタを新たなメモリ割付についてのチェインリスト部分２８６に追加する前に、前回のメモリ割付についてのチェインリスト２８６内のリストからこのポインタを除去することが好ましい。下でさらに述べるように、本発明のこの特徴により、メモリ漏洩を自動的に検出することが可能となる。

【０１８４】結果として、メモリ割付構造体２８０のチェインリスト部分２８６に記録されている情報は、連関する割付されたメモリを現に指示するポインタだけを表示する。

【０１８５】図６に示すように、メモリ割付構造体２８０に含まれる初期化ビットベクトル２８８は、割付されたメモリの各バイト当り１個のビットからなり、連関するメモリの各バイトの初期化状態を維持表示する。本発明の好ましい実施例において、割付されたメモリの領域内の全バイトが初期化され終ると、初期化ビットベクトル２８８は廃棄され、この領域全体の初期化状態を表示するフラグがセットされる。

【０１８６】下でさらに述べるように、割付されたメモリからの読み出しがポインタを用いて行われる都度、割付されたメモリのそれぞれのバイトが初期化されたことを確実にするために、初期化ビットベクトル２８８が評価される。これに加えて、下でさらに述べるように、「コンパイルされたコード」または「翻訳されたコード」のいずれかによって割付されたメモリの初期化が検出された後に初期化ビットベクトル２８８の適切なビットを更新するためのメカニズムが設けられる。

【０１８７】本発明の好ましい実施例においては、「ｍａｌｌｏｃ」関数のようなメモリ割付関数、に連関する「実行後ラッパ関数」が、大抵のプログラミング実例では出会いそうにないような既知の予め定義された値をこの関数によって割付されているメモリの４バイト領域、の各々について事前のマーク付けを行うのが好ましい。この事前マーク付けされた値によって、メモリ領域の初期化の検出が可能となる。

【０１８８】プログラミング環境では１６進値「ＦＦＦＡ　５Ａ５Ａ」には次の２つの理由から通常出会わないことが判っている。第１の理由は、もしこの値が浮動小数点値として用いられる場合この値がＩＥＥＥ浮動小数点基準において「非番号」として定義されるので、浮動小数点トラップエラーが発生するためである。第２の理由は、もしプログラマがこの値をポインタとして用いようとする場合、大抵のマシンにセグメンテーションまたはメモリ欠陥エラーが発生するためである。

【０１８９】すべての動的に割付されたメモリを１６進値「ＦＦＦＡ　５Ａ５Ａ」で事前マーク付けすることに加えて、すべての初期化されていない自動変数に連関するメモリ位置が実行時において同様にマーク付けされる

こともまた好ましい。なお、これに加えて、すべての初期化されていない静的変数は、実行時においてリンク装置４０により「0」にセットされる。

【０１９０】「ｍａｌｌｏｃ」関数のようなメモリ割付関数に通常連関する動作速度は遅いので、多くのプログラマはしばしば、１個の大きなメモリブロックの割付を行い、それからその大きなブロックを、必要に応じてより小さい部分に分解し、割付を受けたこれらのより小さい部分の各々を少なくとも１個のポインタでアクセス可能なようにしている。しかし、上に概略説明したポインタ検査表更新プロセスによれば、各ポインタの有効領域範囲は通常、大きなメモリブロック全体としてポインタ検査表２００に記録されることになる。

【０１９１】すなわち、ポインタ検査表２００は、より大きなメモリ割付分のうちのより小さな部分だけを指示するように意図されているポインタについての適切な領域範囲を表示する必要がある。したがって、より小さい領域範囲を記録するために、ポインタ検査表２００内のエントリにアクセスできることが好ましい。

【０１９２】本発明のさらに別の特徴によれば、割付されたメモリの割付解除（ｆｒｅｅライブラリ関数を呼び出すことによって行われる）も図１２に示す「メモリ割付解除をモニタするためのサブルーチン」によってモニタされる。メモリ割付解除関数が呼び出されるとき、メモリ割付解除関数の引数は一般に、割付されたメモリスペースを指示するポインタのうちの１個である。

【０１９３】共通するプログラミングエラーの１つは、ある１個のポインタを用いてメモリを割付解除し、それから同じメモリスペースを、その同じメモリスペースを指示するように前に定義されていた別個のポインタでアクセスしようと意図する場合である。

【０１９４】連関するコンパイルされたメモリ割付解除関数がステップ７３７において呼び出され実行される前に、「メモリ割付解除をモニタするためのサブルーチン」のステップ７０５からステップ７３５までにおいて「実行後ラッパ関数」が起動されるのが好ましい。「実行後ラッパ関数」は、解除された（ｆｒｅｅされた）メモリにアクセスしようと意図するときに引き続いてエラーが発生するのを防止する。

【０１９５】「メモリ割付解除をモニタするためのサブルーチン」は、メモリ割付解除関数へ送られたポインタ引数について、ステップ７０５においてポインタ検査表２００内の横列を最初に探索する。

【０１９６】ステップ７１０において、ポインタが「解放」（ＦＲＥＥＤ）状態にあることを状態エントリ２６０が表示しているかどうかを定めるテストが行われる。もしステップ７１０において、状態が「解放」であると判断された場合、ステップ７１５においてエラーメッセージが生成される。その理由は、そのポインタによって指示された割付されたメモリが既に解放されているからである。その後、手順はステップ７４０においてサブルーチンから出て、サブルーチンが終了する。

【０１９７】ステップ７２０において、状態エントリ２６０が「メモリ割付済」の状態または「不明」の状態かどうかを定めるテストが、「メモリ割付解除をモニタするためのサブルーチン」によって行われる（状態エントリ２６０（図２）および状態部分２９０（図６）において、「メモリ割付済」の状態および「不明」の状態をそれぞれ「割付済」および「不明」とマーク付けする）。

【０１９８】もしステップ７２０において、状態が「メモリ割付済」の状態または「不明」の状態のいずれでもないと判断された場合、ステップ７２５においてエラーメッセージが生成される。その理由は、これらの状態だけが、割付されたメモリについての有効な状態コードだからである。その後、手順はステップ７４０においてサブルーチンから出て、サブルーチンが終了する。

【０１９９】もしステップ７２０において、状態が「メモリ割付済」の状態または「不明」の状態であると判断された場合、「メモリ割付解除をモニタするためのサブルーチン」がステップ７２７において、下側境界エントリ２５０ａ内でそのポインタによって指示されるメモリ割付構造体２８０にアクセスし、「解放」の状態を表示するように状態部分２９０（図６）をセットする。

【０２００】その後、ステップ７３０において、「メモリ割付解除をモニタするためのサブルーチン」が、下側境界エントリ２５０ａ内でそのポインタによって指示されるメモリ割付構造体２８０にアクセスし、割付されたメモリスペースを指示するすべてのポインタのリストを検索により取り出す。

【０２０１】ステップ７３５において、前のステップにおいて検索により取り出されたリストに表示される各ポインタについて、ポインタ検査表２００の適切な横列がアクセスされ、状態エントリ２６０が「解放」とマーク付けされる。

【０２０２】これに加えて、ステップ７３７におけるコンパイルされたメモリ割付解除関数の実行に続いて、ステップ７３９において、「実行後ラッパ関数」がリストに入れられた各ポインタの実際の内容をゼロ値にセットするのが好ましい。この場合、もしゼロ値を有するこれらのポインタのうちの１個を関連解除する試みがなされると、エラーメッセージが生成される。この後、ステップ７４０において手順はサブルーチンを出て、サブルーチンが終了する。

【０２０３】なお、翻訳された関数においてローカル変数について形成されたメモリスペースを静的ポインタが指示すると、暗黙のメモリ割付解除が生じる。翻訳された関数が戻ると、割付されたメモリスペースは、翻訳器１５によって自動的に割付解除される。

【０２０４】静的ポインタは、関数からの戻りに続いて、無効メモリスペースを指示するので、静的ポインタ

の値をゼロ値にセットすることが好ましく、また、静的ポインタについてのポインタ検査表２００内の横列が、「0」の内容を表示するように更新されるのが好ましい。これは、静的ポインタにローカル変数のアドレスを割り当てた「翻訳された関数」から戻ったときに図１２の「メモリ割付解除をモニタするためのサブルーチン」を実行することによって実現される。

【０２０５】［実行時エラー検査動作］評価器３５が実行時において内部疑似コードの各ノードを評価すると、図１３に示すような、そのノードが、エラー検査ルーチンに関連するコマンドを有するかどうかを定める「実行時エラー検査プロセス」が実現される。「実行時エラー検査プロセス」は、図７を参照して上に述べた「読み出し時エラー検査プロセス」によってパーズ木に挿入されたあるエラー検査コマンドおよび追加情報に応答して、内部疑似コードの各ノードを評価し、実行時エラー検査タスクを実現する。

【０２０６】「実行時エラー検査プロセス」は、図１３に示すように、評価中のノードが実行時エラー検査ルーチンに関連する５個のコマンドのうちの１個を有するかどうかを判断する。もしエラー検査コマンドが検出された場合、「実行時エラー検査プロセス」が、図１３に示すように、適切な応答を起動する。

【０２０７】ステップ８００において手順が「実行時エラー検査プロセス」に入り、ここで内部疑似コードの各ノードが評価された後、もしステップ８１０において「ｄｉｍｃｈｋ」コマンドが存在することが検出されると、図１４を参照して下に述べる「実行時配列次元検査サブルーチン」が実行される。

【０２０８】「実行時配列次元検査サブルーチン」は、実行時において配列が、不正な添字を参照として付けられているかどうかを定める。ステップ９１０（図１４）において、変数配列添字が、実行時条件に基づいて計算される。その後、ステップ９２０において、計算された添字参照が負数かまたは有効最大次元を超えているかどうかを定めるテストが行われる。なお、有効最大次元は、読み出し時において計算され、「ｄｉｍｃｈｋ」コマンドに入れられたものである。

【０２０９】もしステップ９２０において、計算された添字参照が不正値であると判断された場合、ステップ９２４において、有効最大次元が「1」であるかどうかを定めるテストが行われる。

【０２１０】もしステップ９２４において、有効最大次元が「1」であると判断された場合、本明細書に開示されているポインタ検査手段がステップ９２８において用いられて、構造体が全体として有効範囲内にあるかどうかが定められる。単一要素を有する配列で構造体宣言を終り次いで実行時における配列の長さを選択するという共通のプログラミング手法が、このステップ９２４において行われるテストによって可能になる。なお、構造体が割付されるときには、配列の追加構成部分に対して追加のメモリスペースが割付される。

【０２１１】もしステップ９２４において、有効最大次元が「1」ではないと判断された場合、エラーは、この共通プログラミング手法の結果ではない。そして、ステップ９３０においてエラーメッセージが生成される。ステップ９３０の実行後、またはもしステップ９２０において、計算された添字参照が正規の値であると判断された場合、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２１２】ステップ８２０において、評価中の内部疑似コードのノードが「ｓａｖｐｔｒ」コマンドを有するかどうかを定めるテストが、「実行時エラー検査プロセス」によって行われる。上に述べたように、「ｓａｖｐｔｒ」コマンドが「翻訳された原始コード」に挿入されている。したがって、実行時において評価されると、各「ｓａｖｐｔｒ」コマンドによって、ポインタ検査表２００内の適切な横列に含まれるポインタ情報がポインタ保存スタック６５にコピーさせられる。

【０２１３】もしステップ８２０において、ノードが「ｓａｖｐｔｒ」コマンドを有すると判断された場合、ステップ８２５において評価器３５が、コピーされているポインタのアドレスをインタプリタスタック６０の最上部から検索により取り出す。その後、評価器３５が、インタプリタスタック６０から検索により取り出されたアドレスを用いてポインタ検査表２００の適切な横列を探索し、ポインタ情報をポインタ検査表２００の横列からポインタ保存スタック６５にコピーする。

【０２１４】これに加えて、もし、上に述べたように、「ｓａｖｐｔｒ」コマンドが関数を指示するポインタを有する場合、この、関数を指示するポインタはポインタ保存スタック６５にも入れられる。

【０２１５】上に述べたように、ポインタ情報は、図１７に示し下に説明するような「実行時ポインタ検査表更新サブルーチン」のステップ１０６０からステップ１０８０までにおいてポインタ検査表２００に入れるために、評価器３５によってポインタ保存スタック６５から次に検索により取り出される。ステップ８２５の実行後、手順はステップ８００に戻り、上記のように進む。

【０２１６】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが「ｔｂｌｕｐｄ」コマンドを有することがステップ８３０（図１３）において検出された場合、手順はステップ１００２において、図１５から図１７までに示す「実行時ポインタ検査表更新サブルーチン」に入る。

【０２１７】ステップ１００２（図１５）において、「ｔｂｌｕｐｄ」コマンドのノードに含まれる特徴情報が評価され、それからステップ１００４からステップ１０９０（図１７）までにおいて、型のマッチングが得られるまで、いくつもの条件に対して検査される。なお、

割り当てられるポインタのアドレスは、実行時においてインタプリタスタック６０から得られる。割り当てられるポインタについてのポインタ情報で更新する必要のあるポインタ検査表２００内の適切な横列を探索特定するのにポインタのアドレスが用いられる。

【０２１８】ステップ１００４において、識別特定された変数または関数のアドレスがポインタに割り当てられる内部記号表７５内のエントリ、を指示するポインタがノードに含まれるかどうか、を定めるテストが行われる。

【０２１９】もしステップ１００４において、内部記号表７５内のエントリを指示するポインタがノードに含まれると判断された場合、そのポインタは、識別特定された変数または関数のアドレスおよび、もし利用可能なら、識別特定された変数または関数のサイズを検索により取り出すために、ステップ１００６において内部記号表７５の適切なエントリにアクセスするのに用いられる。

【０２２０】ステップ１００８において、内部記号表７５内でサイズ情報が利用可能だったかどうかを定めるテストが行われる。もしステップ１００８において、内部記号表７５内でサイズ情報が利用可能ではなかったと判断された場合、ステップ１０１０において、状態フラグが「不明」（UNKNOWN）にセットされる。

【０２２１】もしステップ１００８において、内部記号表７５内でサイズ情報が利用可能だったと判断された場合、ステップ１０１４において、そのポインタが関数または変数を指示しているかどうかを定めるテストが行われる。

【０２２２】もしステップ１０１４においてポインタが関数を指示していると判断された場合、ステップ１０１６において、状態フラグが「関数指示」（ＰＦＵＮＣ）にセットされる。この場合、関数は始点からしか初期化されないので、関数を指示するポインタについての有効領域範囲はその関数の始点だけとなる。したがって、ステップ１０１７において有効領域範囲が、ステップ１００６において内部記号表から検索により取り出された始点アドレスにセットされる。

【０２２３】もしステップ１０１４においてポインタが変数を指示していると判断された場合、ステップ１０１８において、状態フラグが「境界あり」にセットされる。ステップ１０１９において、ポインタについての有効領域範囲が、内部記号表７５から検索により取り出されたアドレスおよびサイズ情報を用いて計算される。

【０２２４】ステップ１０２０において、ステップ１００６からステップ１０１９までの実行時に割り当てられるポインタについて定義された情報が、割り当てられるポインタについて設定されたポインタ検査表２００の適切な横列に追加される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２２５】もしステップ１００４（図１５）において、内部記号表７５内のエントリを指示するポインタがノードに含まれないと判断された場合、手順はステップ１０２２に進む。

【０２２６】ステップ１０２２においては、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドだけを含むかどうかを定めるテストが行われる。このコマンドは、もしポインタが第２のポインタの内容を割り当てられる場合には、読み出し時においてパーズ木に入れられる。

【０２２７】もしステップ１０２２において、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドだけを含むと判断された場合は、コピーされる第２のポインタのアドレスを検索により取り出すために、評価器３５がインタプリタスタック６０にアクセスするのが好ましい。その後、ステップ１０２６において、評価器３５が、ステップ１０２４においてインタプリタスタック６０から検索により取り出されたアドレスを用いて第２のポインタに対するポインタ検査表２００内の横列を探索する。

【０２２８】それから、第２のポインタについてポインタ検査表２００から検索により取り出された情報が、割り当てられるポインタに対する横列にコピーされる。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２２９】もしステップ１０２２（図１５）において、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドを含まないと判断された場合、手順はステップ１０３０（図１６）へ進む。

【０２３０】ステップ１０３０においては、ノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含むかどうかを定めるテストが行われる。もしステップ１０３０においてノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含むと判断された場合には、ステップ１０３２において評価器３５が、アドレスおよびサイズ情報に基づいてポインタについての有効な領域範囲を計算する。

【０２３１】ステップ１０３４において状態フラグを「境界あり」にセットした後、割り当てられるポインタに対してステップ１０３２およびステップ１０３４の実行中に定義された情報が、ステップ１０３６において、割り当てられるポインタについてのポインタ検査表２００の適切な横列に入れられる。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２３２】もしステップ１０３０においてノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含まないと判断された場合、手順はステップ１０３８に進む。

【０２３３】ステップ１０３８において、ポインタが不正な値を割り当てられことからノードが「不正」（ＩＬ

ＬＥＧＡＬ）の状態表示を含むかどうかを定めるためのテストが行われる。

【０２３４】もしステップ１０３８において、ノードが「不正」の状態表示を含むと判断された場合、ステップ１０４０において状態フラグが「不正」にセットされる。そして、ポインタについて定義された情報がステップ１０４２において、ポインタ検査表２００の適切な横列に追加される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２３５】もしステップ１０３８において、ノードが「不正」の状態表示を含まないと判断された場合、手順はステップ１０４４に進む。

【０２３６】ステップ１０４４においては、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドを構造体メンバのサイズとともに含むかどうかを定めるテストが行われる。もしステップ１０４４において、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドを構造体メンバのサイズとともに含むと判断された場合、ステップ１０４６において評価器３５が、割り当てられる構造体メンバのアドレスを検索により取り出す。

【０２３７】なお、「ｒｅａｄｉｎｔｓｔｋ」コマンドは、翻訳された原始コードに、読み出し時中に位置を占めており、その位置占めは、構造体メンバのアドレスがインタプリタスタック６０の最上部にある時点にこのコマンドが評価されることが可能となるような仕方でなされる。

【０２３８】ステップ１０４８において評価器３５が、サイズ情報および検索により取り出されたアドレス情報に基づいて、ポインタについての有効領域範囲を計算し、それからステップ１０５０において、状態フラグを「境界あり」にセットする。ステップ１０５２において、割り当てられるポインタについて定義された情報がポインタ検査表２００の適切な横列に追加される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２３９】もしステップ１０４４において、ノードが「ｒｅａｄｉｎｔｓｔｋ」コマンドを構造体メンバのサイズとともに含まないと判断された場合、手順はステップ１０６０（図１７）に進む。

【０２４０】ステップ１０６０においては、ノードが「ｒｄｐｔｒｓｔｋ」コマンドを含むかどうかを定めるテストが行われる。もしステップ１０６０において、ノードが「ｒｄｐｔｒｓｔｋ」コマンドを含むと判断された場合、ステップ１０６４において評価器３５が、ポインタ保存スタック６５の内容を検索により取り出す。

【０２４１】上に述べたように、「ｒｄｐｔｒｓｔｋ」コマンドが検出されたときには、連関する「ｓａｖｐｔｒ」コマンドがステップ８２５（図１３）において既に評価されており、これによってポインタ検査表２００を更新するために必要な適切なポインタ情報がポインタ保存スタック６５に入れられた。

【０２４２】ステップ１０６６においては、ポインタ保存スタックが、関数を指示するポインタを含むかどうかを定めるテストが行われる。このテストは、エラー検査タスクを翻訳された原始コードとコンパイルされたライブラリ関数とから構成されるソフトウェアプログラム上でも稼動可能にするために実現される。

【０２４３】なお、もしポインタ保存スタック６５が、呼び出された関数へポインタ情報を送るために（ステップ５４５およびステップ５４７）、または呼び出された関数からポインタ情報を返すために（ステップ５５０およびステップ５５２）用いられる場合には、関数を指示するポインタは、読み出し時において「ｓａｖｐｔｒ」コマンドに含まれる。

【０２４４】関数を指示するポインタは、呼び出された関数へポインタ情報を送る場合はにポインタ保存スタック６５からポインタ情報を受け取るべき関数を識別特定し、または、呼び出し側関数へポインタ情報を返す場合にはポインタ保存スタック６５にポインタ情報を入れた関数を識別特定する。

【０２４５】もしステップ１０６６において、ポインタ保存スタックが、関数を指示するポインタを含まないと判断された場合、ポインタ保存スタック６５におけるポインタ情報の有効性をテストする必要はなく、手順は下に述べるステップ１０８０に進む。

【０２４６】もしステップ１０６６において、ポインタ保存スタックが、関数を指示するポインタを含むと判断された場合には、関数を指示するポインタの有効性をテストする必要がある。すなわちステップ１０６８において、ポインタ保存スタック６５が、呼び出された関数へポインタ情報を送るために用いられているかまたは呼び出された関数からポインタ情報を返すために用いられているかを定めるためのテストが行われる。

【０２４７】もしステップ１０６８において、ポインタ保存スタック６５が、呼び出された関数へポインタ情報を送るために用いられていると判断された場合、送られたポインタ情報が、呼び出された関数によって検索により取り出せるように呼び出し側関数によってポインタ保存スタック６５に入れられたことを確認するために、ステップ１０７０において、検索により取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しいかどうかがを定めるテストが行われる。

【０２４８】もしステップ１０７０において、検索により取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しくないと判断された場合、ポインタ保存スタック６５からのポインタ情報は、ポインタ検査表２００に入れるべきではない。そして手順はステップ８００（図１３）において、「実行時エラー検査プロセス」に戻る。

【０２４９】もしステップ１０７０において、検索によ

り取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しいと判断された場合、ポインタ保存スタック６５からのポインタ情報は、ステップ１０８０においてポインタ検査表２００に入れるべきである。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２５０】もしステップ１０６８において、ポインタ保存スタック６５が、呼び出し側関数へポインタ情報を返すために用いられていると判断された場合、ポインタ情報が、呼び出し側関数によって検索して取り出せるように呼び出された関数によってポインタ保存スタック６５に入れられたことを確認するために、ステップ１０７４において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレス、すなわち呼び出された関数のアドレスに等しいかどうかがを定めるテストが行われる。

【０２５１】もしステップ１０７４において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレスに等しくないと判断された場合、ポインタ保存スタック６５からのポインタ情報は、ポインタ検査表２００に入れるべきではない。そして手順はステップ８００（図１３）において、「実行時エラー検査プロセス」に戻る。

【０２５２】もしステップ１０７４において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレスに等しいと判断された場合、ポインタ保存スタック６５からのポインタ情報は、ステップ１０８０においてポインタ検査表２００に入れるべきである。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２５３】ステップ１０８０においてポインタ検査表２００が、ステップ１０６４においてポインタ保存スタック６５から検索により取り出されたポインタ情報を用いて更新される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２５４】もしステップ１０６０において、ノードが「ｒｄｐｔｒｓｔｋ」コマンドを含まないと判断された場合、手順はステップ１０９０に進む。

【０２５５】ステップ１０９０においては、ノードが「不明」の状態表示を含むかどうかを定めるテストが行われる。もしステップ１０９０において、ノードが「不明」の状態表示を含むと判断された場合、ステップ１０９２において評価器３５が、状態フラグを「不明」にセットし、それから、割り当てられるポインタについて定義された情報がステップ１０９４において、ポインタ検査表２００の適切な横列に追加される。

【０２５６】その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２５７】もしステップ１０９０（図１７）において、ノードが「不明」の状態表示を含まないと判断された場合、手順はステップ１０９６に進む。

【０２５８】もし手順がステップ１０９６に到達した場合、「ｔｂｌｕｐｄ」ノードは、上に設定されたテストのうちのどれに基づく特徴付けもできないことになる。したがって、ステップ１０９６においてエラーメッセージが生成され、その後、手順はステップ１０９８において「実行時ポインタ検査表更新サブルーチン」から出て、サブルーチンが終了する。

【０２５９】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが「ｐｔｒｃｈｋ」コマンドを有することがステップ８４０（図１３）において検出された場合、手順はステップ１１０４において、図１８および図１９に示す「実行時ポインタ検査サブルーチン」に入る。上に述べたように、「実行時ポインタ検査サブルーチン」は、関連解除される各ポインタの有効性のテストを行う。

【０２６０】ステップ１１０４において、検査されるポインタのアドレスを検索により取り出すために評価器３５が、インタプリタスタック６０にアクセスする。上に述べたように、「ｐｔｒｃｈｋ」コマンドは、実行時において評価されているとき、検査されているポインタのアドレスがインタプリタスタック６０の最上部に来るような仕方で、パーズ木の中に位置させる。

【０２６１】ステップ１１０８において評価器３５が、ステップ１１０４においてインタプリタスタック６０から検索により取り出されたポインタアドレスを利用して、検査されているポインタに対するポインタ検査表２００内の横列を探索する。

【０２６２】ステップ１１１２において、検査されているポインタに対するポインタ検査表２００内の状態エントリ２６０が「ＰＦＵＮＦ」にセットされているかどうかを定めるためのテストが行われる。もしステップ１１１２において、状態エントリ２６０が「関数指示」にセットされていると判断された場合、ステップ１１１６において関数のアドレスが内部記号表７５から検索により取り出される。

【０２６３】ステップ１１２０においてポインタ検査表２００内のポインタ内容エントリ２４０が、内部記号表７５から検索により取り出されたアドレスによって書き直される。このルーチンは、翻訳器１５にロードされた最新版の関数が実行されることを確実にするために実現される。その後、手順はステップ１１２４に進む。

【０２６４】ステップ１１２４において、ポインタ検査表２００内のポインタ内容エントリ２４０に記録されている値が、ポインタの実際の内容に等しいかどうかを定めるためのテストが行われるのが好ましい。この好ましい実施例においては、評価器３５が知らずに、「コンパイルされたコード」によって行われたポインタ修正を検出するために、上記ラッパ関数実現例に追補されるメカニズムが設けられる。評価器３５が前の修正を知らなか

ったので、ポインタ検査表２００は適切な更新が行われなかった。

【０２６５】もしステップ１１２４において、これらの値が等しくないと判断された場合、このポインタについて前に検出されなかった修正が発生していることになる。この修正は「コンパイルされたコード」によって有効正当に行われたものと仮定する。

【０２６６】そして、ステップ１１２８において、検査されるポインタについて状態フラグが「不明」にセットされ、それ以上の追加検査は行われない。このようにして、「コンパイルされたコード」による有効なポインタ修正についての誤った警告が防止される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２６７】もしステップ１１２４において、記録されている内容がポインタの実際の内容に等しいと判断された場合、手順はステップ１１３２に進む。

【０２６８】ステップ１１３２において、検査されるポインタについてポインタ検査表２００内の状態エントリ２６０が「不正」とセットされているかどうかを定めるためのテストが行われる。もしステップ１１３２において、状態が「不正」とセットされていると判断された場合、一定ゼロ値または負数のポインタが関連解除されるというエラーが今発生していることになる。その結果、手順は、図２２を参照して下に述べる診断サブルーチンに進む。

【０２６９】もしステップ１１３２において、状態が「不正」とセットされていないと判断された場合、手順はステップ１１４０（図１９）に進む。

【０２７０】ステップ１１４０（図１９）においては、検査されるポインタについてポインタ検査表２００内の状態エントリ２６０が「解放」とセットされているかどうかを定めるためのテストが行われる。

【０２７１】もしステップ１１４０において、状態が「解放」にセットされていると判断された場合には、割付を解除されていたメモリがアクセスされているという不適切な状態、すなわち解放メモリアクセスエラーが発生していることになる。その結果、手順は、図２２を参照して下に述べる診断サブルーチンに進む。

【０２７２】もしステップ１１４０において、状態が「解放」にセットされていないと判断された場合には、手順はステップ１１４８に進む。

【０２７３】ステップ１１４８において、検査されるポインタについてポインタ検査表２００内の状態エントリ２６０が「境界あり」とセットされているかどうかを定めるためのテストが行われる。

【０２７４】もしステップ１１４８において、状態が「境界あり」にセットされていると判断された場合には、ステップ１１５２において、検査されているポインタの実際の内容が、検査されているポインタについてのポインタ検査表２００の有効メモリ境界エントリ２５０ａ、２５０ｂに定められるような有効領域範囲内にあるかどうかを定めるためのテストが行われる。

【０２７５】もしステップ１１５２において、検査されているポインタの、検索により取り出されたポインタ値が、有効領域範囲内にあると判断された場合、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２７６】しかし、もしステップ１１５２において、検索により取り出されたポインタ値が有効領域範囲内にないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図２２を参照して下に述べる診断サブルーチンに進む。

【０２７７】もしステップ１１４８において、状態が「境界あり」にセットされていないと判断された場合には、手順はステップ１１６０に進む。

【０２７８】ステップ１１６０においては、検査されるポインタについてポインタ検査表２００内の状態エントリ２６０が「割付済」にセットされているかどうかを定めるためのテストが行われる。

【０２７９】もしステップ１１６０において、状態が「割付済」にセットされていると判断された場合には、ステップ１１６２において、ポインタの内容が、図６に示す連関するメモリ割付構造体２８０の下側および上側境界エントリ２８２、２８４によって定義されるような割付されたメモリの有効な領域範囲の中にあるかどうかを定めるためのテストが行われる。

【０２８０】もしステップ１１６２において、ポインタの内容が、割付されたメモリについての有効な領域範囲の中にないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図２２を参照して下に述べる診断サブルーチンに進む。

【０２８１】もしステップ１１６２において、ポインタの内容が、割付されたメモリについての有効な領域範囲の中にあると判断された場合には、ステップ１１６４において、ポインタが、割付されたメモリに対して読み出しまたは書き込みを行うために関連解除されるかどうかを定めるためのテストが行われる。

【０２８２】もしステップ１１６４において、ポインタが、割付されたメモリに対して書き込みを行うために関連解除されると判断された場合、上に述べた初期化ビットベクトル２８８の適切なビットが、ステップ１１６８において、割付されたメモリのうちの対応するバイトの新たな初期化状態を表示するように更新される。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２８３】もしステップ１１６４において、ポインタが、割付されたメモリに対して読み出しを行うために関連解除されると判断された場合、ステップ１１７２において、割付されたメモリのうちの対応するバイトが初期

化されているかどうかを定めるために、初期化ビットベクトル２８８の適切なビットがアクセスされる。

【０２８４】ステップ１１７４において、読み出しが行われる割付されたバイトが初期化されているかどうかを定めるためのテストが行われる。もしステップ１１７４において、これらのバイトが初期化されていないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図２２を参照して下に述べる診断サブルーチンに進む。

【０２８５】もしステップ１１７４において、これらのバイトが初期化されていると判断された場合には、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０２８６】もしステップ１１６０において、状態が「割付済」にセットされていないと判断された場合には、手順はステップ１１７６に進む。もし手順がステップ１１７６に到達した場合、状態エントリ２６０（図２）に記録されている情報は、上に設定されたテストのうちのどれに基づく特徴付けもできないことになる。したがって、ステップ１１７６においてエラーメッセージが生成され、その後、手順は「実行時ポインタ検査サブルーチン」から出て、サブルーチンが終了する。

【０２８７】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが、「コンパイルされた関数」についての「ｃａｌｌ」（呼び出し）コマンドを有することがステップ８５０（図１３）において検出された場合、手順はステップ１２０２において、図２０に示す「「コンパイルされた関数」エラー検査プロセス」に入る。

【０２８８】もし呼び出された「コンパイルされた関数」が下にさらに述べるように割付されたメモリ領域へのアクセスを有する場合に、「「コンパイルされた関数」エラー検査プロセス」によって、与えられた「コンパイルされた関数」に連関する「ラッパ関数」を起動するためおよび図２１に示す初期化ビットベクトル維持サブルーチンを起動するためのメカニズムが得られる。

【０２８９】ステップ１２０２において、呼び出された「コンパイルされた関数」が、連関する「実行前ラッパ関数」を有するかどうかを定めるためのテストが、「「コンパイルされた関数」エラー検査プロセス」によって行われる。

【０２９０】もしステップ１２０２において、呼び出された「コンパイルされた関数」が、連関する「実行前ラッパ関数」を有すると判断された場合、上に述べたような、必要なポインタ検査およびその他の関数を行うために、ステップ１２０４において、「実行前ラッパ関数」が実行される。「実行前ラッパ関数」は、連関する「コンパイルされた目的コード関数」と同じ引数を送られる。その後、手順はステップ１２０５に進む。

【０２９１】もしステップ１２０２において、呼び出された「コンパイルされた関数」が、連関する「実行前ラッパ関数」を有しないと判断された場合、手順はステップ１２０５に進む。

【０２９２】ステップ１２０５においては、割付アクセスフラグが、呼び出された「コンパイルされた関数」に対応する内部記号表７５内のエントリにセットされているかどうかを定めるためのテストが行われる。割付アクセスフラグは、もしこの関数が割付されたメモリのどれかにアクセスを有する場合にステップ３３７において読み出し時にセットされた。

【０２９３】もしステップ１２０５において、割付アクセスフラグがセットされていると判断された場合、ステップ１２０７において、図２１に示す「初期化ビットベクトル維持サブルーチン」が実行される。手順がステップ１２１０においてこのサブルーチンに入ることで開始されるこの「初期化ビットベクトル維持サブルーチン」によって、翻訳器１５によっては通常に検出されない割付されたメモリの初期化の「コンパイルされた関数」による検出のためのメカニズムが得られる。

【０２９４】なお、割付されたメモリの「翻訳された関数」による初期化は、図１９に示す「実行時ポインタ検査サブルーチン」のステップ１１６４およびステップ１１６８の実行時に検出される。このようにして、初期化ビットベクトル２８８は、初期化が「コンパイルされた関数」または「翻訳された関数」のいずれによって行われたかに関係なく、割付されたメモリ領域の各対応するバイトの初期化状態を適切に記録する。

【０２９５】上に述べたように、本発明の好ましい実施例においては、メモリ初期化の検出を容易にするために、初期化されていない自動変数および割付されたメモリ領域がすべて、例えば「ＦＦＦＡ　５Ａ５Ａ」のような既知の４バイトの１６進語」によって事前マーク付けされる。

【０２９６】ステップ１２１０（２１図）において「コンパイルされた関数」が呼び出された時点に、初期化されていない対応するバイトのすべてを識別特定するために、評価器３５が、「コンパイルされた関数」によってアクセスされる割付されたバイトに対応する初期化ビットベクトル２８８（図６）の各ビットの解析を行う。

【０２９７】それから、「コンパイルされたコード」によってアクセスされる初期化されていない対応するバイトのすべてを識別特定するために、ステップ１２２０において評価器３５が、巡回冗長テスト（ＣＲＣテスト）を行う。ステップ１２２５において、「コンパイルされた関数」が適切な引数で呼び出され、実行される。

【０２９８】「コンパイルされたコード」の実行に続いて、手順はステップ１２３０に進み、ここで、「コンパイルされたコード」の実行前には「コンパイルされたコード」がアクセスでき初期化されていなかったバイトのすべてに対して第２のＣＲＣテストが行われる。

51

【０２９９】次にステップ１２３５において、ＣＲＣテストによってバイトの値が変化したものがあるかどうか、ステップ１２２０において行われたＣＲＣテストの結果が、ステップ１２３０において行われたＣＲＣテストの結果に等しいかどうか、を定めるためのテストが行われる。もしステップ１２３５において、これら２つのＣＲＣテスト値が等しくないと判断された場合には、手順はステップ１２４０に進む。

【０３００】ステップ１２４０においては、割付されたメモリのブロック全体を「初期化済」とマーク付けすべきかどうか、または初期化されたブロック内のバイトだけを「初期化済」とマーク付けすべきかについて定めるために、ユーザによって定義されたフラグ（ユーザ定義フラグ）が解析される。ユーザ定義フラグは、各ソフトウェアデバッグ処理プロセスの開始時にユーザによってセットされるのが好ましい。

【０３０１】このユーザ定義フラグによって、ユーザは、「コンパイルされた関数」がメモリを既知の事前マークされた１６進値で初期化する結果、すなわちメモリを１６進値「ＦＦＦＡ　５Ａ５Ａ」で初期化する結果から来る誤ったエラー発生警告を防止することが可能となる。

【０３０２】次にステップ１２４５において、割付されたメモリのブロック全体を、初期化状態（ただ１個のバイトしか「コンパイルされたコード」によって初期化されなかったような初期化状態も含む）としてマーク付けすべきことを表すようにユーザ定義フラグがセットされているかどうか、または代わりに、初期化されていると判断されたバイトだけを初期化状態としてマーク付けすべきであると指定されているかどうか、を定めるためのテストが行われる。

【０３０３】もしステップ１２４５において、ユーザが、割付されたメモリブロック全体を初期化状態としてマーク付けすべきことを指定している、と判断された場合には、ステップ１２５０において、初期化ビットベクトル２８８内の各ビットが、初期化状態を表すように更新される。

【０３０４】もしステップ１２４５において、ユーザが、実際に初期化されているバイトだけを初期化状態としてマーク付けすべきことを指定している、と判断された場合には、手順はステップ１２５５に進む。

【０３０５】このステップ１２５５においては、割付されたメモリ領域において「ＦＦＡ５Ａ５Ａ」以外の内容を有する４バイトブロックの各々、すなわちもはや既知の事前マーク値を持たないバイト、が識別特定される。その後、ステップ１２５８において、初期化ビットベクトル２８８内の対応するビットが、初期化状態を表すように更新される。

【０３０６】代わりの実施例においては、「コンパイルされた関数」がアクセスする初期化されていないメモリ

52

のすべてのバイトが、ステップ１２２５における「コンパイルされたコード」の実行に先立ってセーブ（保存）される。

【０３０７】この場合、もしステップ１２３５において、「コンパイルされたコード」が割付されたメモリを初期化していると判断されたならば、ステップ１２５５において、初期化されておらず且つアクセス可能なバイトの内容を「コンパイルされた目的コード」の実行に先立って記憶セーブされた値と比較することによって、初期化されたバイトが識別特定される。この実施例においては、割付されたメモリ領域を既知の値、すなわち１６進値「ＦＦＦＡ　５Ａ５Ａ」で事前マーク付けする必要はない。

【０３０８】ステップ１２５０またはステップ１２５８の実行後、またはもしステップ１２３５において、「コンパイルされたコード」が、割付されたメモリのどのバイトも初期化していないと判断された場合、手順はステップ１２６５（図２０）において「「コンパイルされた関数」エラー検査プロセス」に戻る。

【０３０９】もしステップ１２０５（図２０）において、割付アクセスフラグがセットされていないと判断された場合、呼び出された「コンパイルされた関数」は、どの割付されたメモリにもアクセスできない。また「初期化ビットベクトル維持サブルーチン」を実行する必要がない。

【０３１０】したがって、もしフラグがセットされていない場合には、「コンパイルされた目的コード」が適切な引数で呼び出され、実行される。「コンパイルされた目的コード」が実行された後、手順はステップ１２６５（図２０）において「「コンパイルされた関数」エラー検査プロセス」に戻る。

【０３１１】ステップ１２６５においては、呼び出された「コンパイルされた関数」が連関する「実行後ラッパ関数」を有するかどうかを定めるためのテストが、「「コンパイルされた関数」エラー検査プロセス」によって行われる。

【０３１２】もしステップ１２６５において、呼び出された「コンパイルされた関数」が連関する「実行後ラッパ関数」を有すると判断された場合、ステップ１２７０において、「実行後ラッパ関数」が、その連関する「コンパイルされた目的コード関数」によって返された値で実行される。

【０３１３】「実行後ラッパ関数」は、返されたまたは形成されたポインタについて必要なポインタ検査表更新タスク、および上に述べたようなその他必要なタスクを行う。その後、手順はステップ８００（図１３）において「実行時エラー検査プロセス」に戻る。

【０３１４】もし図１８および図１９に示す「実行時ポインタ検査サブルーチン」の実行中にエラーが検出された場合、すなわちステップ１１３２、１１４０、１１５

2、1162、または1174のテスト条件が不成功だった場合、手順は、図22に示す診断サブルーチンに進む。

【0315】手順がステップ1310において「診断サブルーチン」に入ると、エラーメッセージが生成される。このエラーメッセージは、診断情報、すなわちメモリアクセスエラーの種類、およびポインタ検査表200のファイル／ライン番号エントリ270から取り出された、ポインタに最後に修正が行われたライン番号、からなる。

【0316】その後、プログラマは、検出されたエラーを訂正するためにステップ1320において、上に述べたように読み出し期間の手段にアクセスできる。プログラマによるエラー訂正が終ると、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0317】［メモリ漏洩の特定および検出されていないポインタ修正のソースの特定］本発明の別の特徴によれば、メモリ漏洩、すなわち割付されたがもはやアクセス不可能なメモリスペースが、自動的に、またはユーザがメモリ漏洩検出アルゴリズムを起動することによって、検出される。

【0318】メモリ漏洩は一般に、割付されたメモリの第1のブロックを指示するポインタが、第1のブロックについての割付解除なしに、割付されたメモリの第2のブロックを指示するように再度割り当てられるときに生じる。周知のように、メモリ漏洩は全体の動作性能の漸増的な劣化をもたらす。

【0319】上に述べたように、「メモリ割付関数」によるメモリブロックの割付がされると、連関するメモリ割付構造体280（図6）がその連関する「実行後ラッパ関数」によって形成されることが好ましい。メモリ割付構造体280は、上に述べたような、割付されたメモリに関するある情報（例えば割付されたメモリのの中の連関するブロックを現に指示するすべてのポインタのチェインリスト部分286内のリスト）を記録する。

【0320】上に述べたように、新たなポインタが、割付されたメモリを指示するように割り当てられる都度、その新たなポインタはチェインリストに追加される。同様に、割付されたメモリの第1のブロックを前に指示したポインタが、割付されたメモリの第2のブロックを指示するように再割当される都度、第2のブロックに連関するチェインリストに追加される前に、第1のブロックに連関するチェインリストから除去されることが好ましい。

【0321】したがって、メモリ割付構造体280のチェインリスト部分286に記録されている情報は、割り付けられたメモリのうちの連関するブロックを現に指示するポインタだけを表示する。そして、もしチェインリスト部分286の内容が空である場合には、割り付けられたメモリを指示するポインタで、割り付けられたメモリにアクセスするために用いられるようなポインタは存在しない。すなわち、メモリ漏洩が発生しているわけで、エラーメッセージを生成する必要がある。

【0322】上に述べたように、ポインタが、「コンパイルされたコード」によって、評価器35に知られずに、修正される場合を検出するための、ステップ1124およびステップ1128における追補のメカニズムが、図18および図19に示す「実行時ポインタ検査サブルーチン」によって得られる。

【0323】評価器35が、「コンパイルされたコード」によって行われた前の修正を知らなかったので、ポインタ検査表200はこの修正の時点で適切な更新が行われなかった。追補のメカニズムは、検査されるポインタについての状態フラグを「不明」状態にセットし、それ以上のポインタ追加検査は行わない。

【0324】このルーチンは「コンパイルされたコード」による有効なポインタ修正についての誤った警告を防止はするが、修正値がポインタ検査表200に入れられるように、「コンパイルされたコード」による各修正を検出する方が好ましい。

【0325】したがって、「不明」の状態表示を有するすべてのポインタを識別特定するために、ポインタ検査表200の各横列の状態エントリ260をユーザがサーチできるようなメカニズムが好ましい。

【0326】「不明」の表示を有する各ポインタについて、それぞれのポインタを識別特定する診断メッセージが、ファイル／ライン番号エントリ270からの情報とともに生成される。ライン番号情報は、ポインタが最後に更新された時点を表示する。このことから、プログラマは診断情報を用いて、翻訳器が知らなくても、ポインタ値を修正した「コンパイルされたコード」を特定することが可能となる。

【0327】その後、上に述べたような方法で、「コンパイルされたコード」に対する「実行後ラッパ関数」が書かれ、これによって、「不明」状態がさらに発生するのを防止するために、「コンパイルされた関数」の実行に続いて新たなポインタ情報がポインタ検査表200に追加される。

【0328】プログラム開発中にプログラマは、CINリセットコマンドのようなリセットコマンドを起動することによって翻訳器15を頻繁にリセットすることを望んでいる。周知のように、このコマンドは翻訳器15に、読み出しプロセスの終了時に元あった状態と同じ状態に戻るように命令する。実際上、このことは、すべてのデータがその当初の値に再初期化され、「BSS」データが「0」に割り当てられる結果となる。

【0329】ポインタ検査表200は、CINリセットコマンド実行終了時に、読み出しプロセスの終了時にポインタ検査表200があった状態と同じ状態に戻ること

55

が好ましい。したがって、ポインタ検査表２００を再初期化し、それから、上に述べたように、すなわち他の静的データを指示するように初期化された静的ポインタおよび主関数に送られた「ａｒｇｖ」ポインタ引数について、内部疑似コードの実行に先立ってポインタ検査表２００にプレロードされた初期化条件を再ロードする必要がある。

【０３３０】以上の説明は、本発明の一実施例に関するもので、この技術分野の当業者であれば、本発明の種々の変形例を考え得るが、それらはいずれも本発明の技術的範囲に包含される。

【０３３１】

【発明の効果】以上述べたごとく、本発明によれば、「翻訳された原始コード」と「コンパイルされた目的コード」との両方を実行しながらエラー検出タスクを行うことのできるソフトウェアテストおよびデバッグ処理ツールが得られる。また、本発明によれば、与えられたポインタによって指示されたメモリがそれぞれのポインタについての適切な境界範囲内にあることを確実にするソフトウェアテストおよびデバッグ処理が可能となる。

【０３３２】さらに、読み出し時またはパーズ時に導出される情報を利用することにより、実行時に行われる互いに重複するポインタ検査の数を減少させて、より効率的なソフトウェアテストおよびデバッグ処理を行うことができる。

【図面の簡単な説明】

【図１】本発明に基づくメモリアクセスエラー検出システムを示す概略ブロック図である。

【図２】図３の原始コードファイルの例に宣言されている各ポインタについての現ポインタ情報を維持するポインタ検査表である。

【図３】図２のポインタ検査表に列記されるポインタを宣言し初期化する原始コードファイルの例を示す説明図である。

【図４】図３の原始コードファイルのライン１０からライン６０までの実行後の、いくつかのポインタと、それらのポインタによって指示されるそれぞれのメモリスペースとの間の関係を示す説明図である。

【図５】図３の原始コードファイルのライン７０からライン１００までの実行後の、いくつかのポインタと、それらのポインタによって指示されるそれぞれのメモリスペースとの間の関係を示す説明図である。

【図６】図３の原始コードファイルのライン２００からライン２１０までの実行後の、ポインタ「ｐｔｒ＿ａｌｌｏｃ」と、このポインタによって指示される割付されたメモリブロックとの間の関係、およびこの割付されたメモリブロックについての追加情報を維持する連関メモリ割付構造体を示す説明図である。

【図７】パーズ木（構文木）を解析する際にメモリエラー検出コード挿入器によって用いられるような読み出し時エラー検査プロセスの例を示す流れ図である。

56

【図８】図７のエラー検査プロセスによって用いられるような読み出し時配列次元検査サブルーチンの例を示す流れ図である。

【図９】図７のエラー検査プロセスによって用いられるような読み出し時ポインタ表更新サブルーチンの例を、図１０と一体で示す流れ図である。

【図１０】図７のエラー検査プロセスによって用いられるような読み出し時ポインタ表更新サブルーチンの例を、図９と一体で示す流れ図である。

【図１１】図７のエラー検査プロセスによって用いられるような読み出し時ポインタ検査サブルーチンの例を、図９と一体で示す流れ図である。

【図１２】コンパイルされたメモリ割付解除関数を実行しながら評価器が用いるようなメモリ割付解除モニタ処理サブルーチンの例を示す流れ図である。

【図１３】実行時において内部疑似コードのノードを解析しながら評価器によって用いられるような実行時エラー検査プロセスの例を示す流れ図である。

【図１４】図１３のエラー検査プロセスによって用いられるような実行時配列次元検査サブルーチンの例を示す流れ図である。

【図１５】図１３のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図１６および図１７と一体で示す流れ図である。

【図１６】図１３のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図１５および図１７と一体で示す流れ図である。

【図１７】図１３のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図１５および図１６と一体で示す流れ図である。

【図１８】図１３のエラー検査プロセスによって用いられるような実行時ポインタ検査サブルーチンの例を、図１９と一体で示す流れ図である。

【図１９】図１３のエラー検査プロセスによって用いられるような実行時ポインタ検査サブルーチンの例を、図１８と一体で示す流れ図である。

【図２０】図１３のエラー検査プロセスによって用いられるような「「コンパイルされた関数」エラー検査プロセス」の例を示す流れ図である。

【図２１】図２０の「「コンパイルされた関数」エラー検査プロセス」によって用いられるような初期化ビットベクトル維持サブルーチンの例を示す流れ図である。

【図２２】図１８および図１９の実行時ポインタ検査サブルーチンによって用いられるような診断サブルーチンの例を示す流れ図である。

【図２３】ポインタ「ｐｔｒ＿ｐａｒｔ」を宣言し初期化する原始コードファイルの例を示す説明図である。

【図２４】図２３に示す原始コードファイルのデータ流れ解析中に設定される流れセット例を示す説明図であ

57

る。

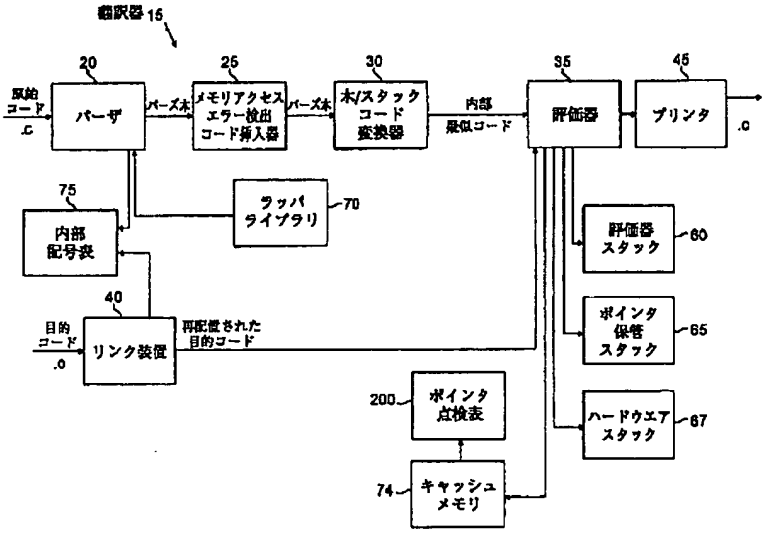
【符号の説明】

１５　翻訳器（インタプリタ）
２０　パーザ（構文解析器）
２５　メモリアクセスエラー検出コード挿入器
３０　木／スタックコード変換器
３５　評価器
４０　リンク装置
４５　プリンタ
６０　インタプリタスタック
６５　ポインタ保存スタック
６７　ハードウェアスタック
７０　ラッパライブラリ
７４　キャッシュメモリ

58

７５　内部記号表
２００　ポインタ検査表
２２０、２２２、２２４　横列
２３０　ポインタアドレスエントリ
２４０　ポインタ内容エントリ
２５０ａ、２５０ｂ　有効メモリ境界エントリ
２６０　状態エントリ
２７０　ファイル／ライン番号エントリ
２８０　メモリ割付構造体
２８２　下側境界部分
２８４　上側境界部分
２８６　チェインリスト部分
２８８　初期化ビットベクトル
２９０　状態部分

【図１】

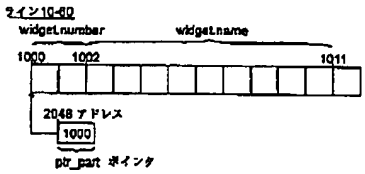


【図２】

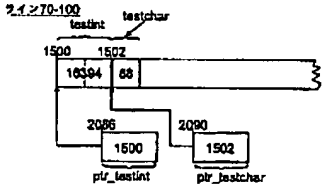
ポインタ点検表 200

	ポインタ	ポインタアドレス	ポインタ内容	有効メモリ境界 下側	有効メモリ境界 上側	状態	ファイル/ライン番号
横列							
220	ptr_part	2048	1000	1000	1011	境界あり	テストファイル 160
222	ptr_testint	2088	1500	1500	1501	境界あり	テストファイル 190
	⋮						
	ptr_alloc	2200	3000	3500		割付済	テストファイル 210
	⋮						
224	ptr_testchar	2090	1502	1502	1502	境界あり	テストファイル 100
		230 エントリ	240 エントリ	250a エントリ	250b エントリ	260 エントリ	270 エントリ

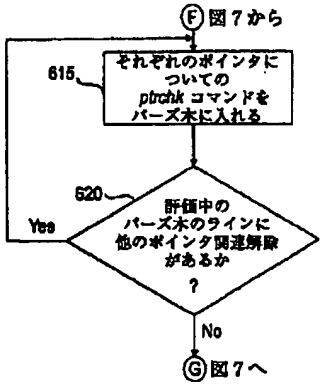
【図４】



【図５】



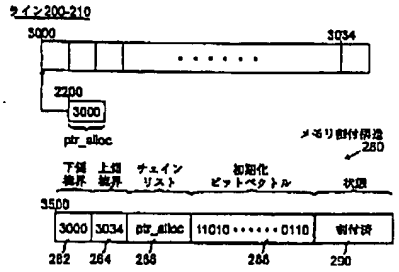
【図１１】



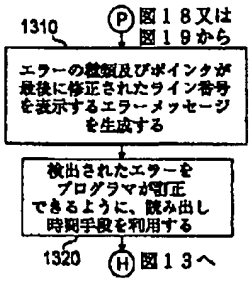
【図3】

```
原始コードファイル=「テストファイル」
      .
      .
      .
I10     struct part {
I20     int number;
I30     char name [10];
I40     } widget;
I50     struct part *ptr_part;
I60     ptr_part = &widget;
      .
      .
      .
I70     int testint = 16394;
I80     char testchar = 88;
I90     int *ptr_testint = &testint;
I100    char *ptr_testchar = &testchar;
      .
      .
      .
I200    char *ptr_alloc;
I210    ptr_alloc = malloc (35 *sizeof (char));
      .
      .
      .
I300    ptr_testchar = &widget.name[0];
I310    *ptr_testint = 12024;
      .
      .
      .
```

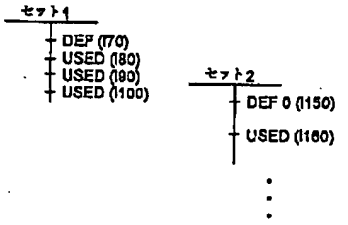
【図6】



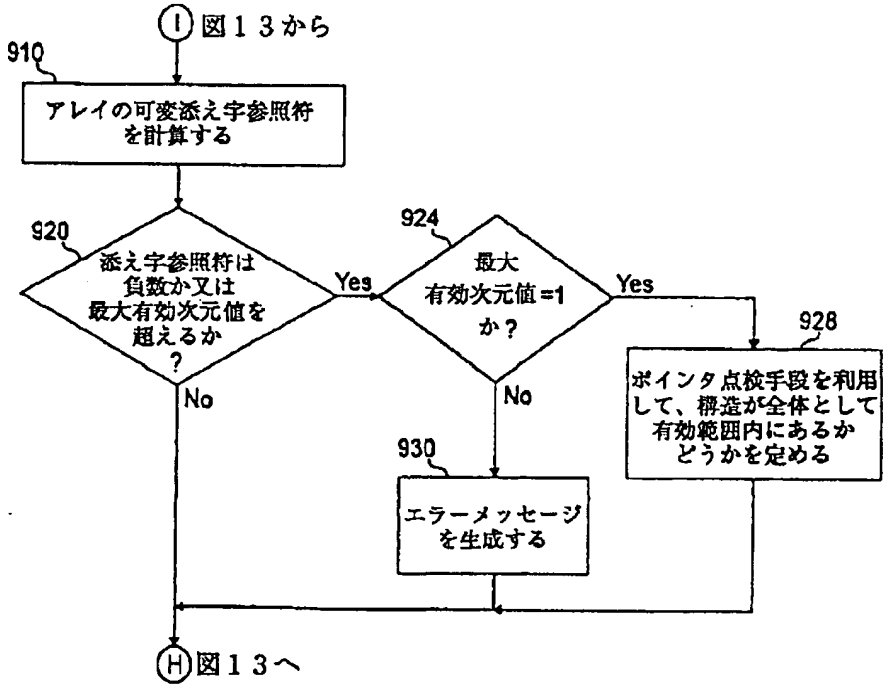
【図22】



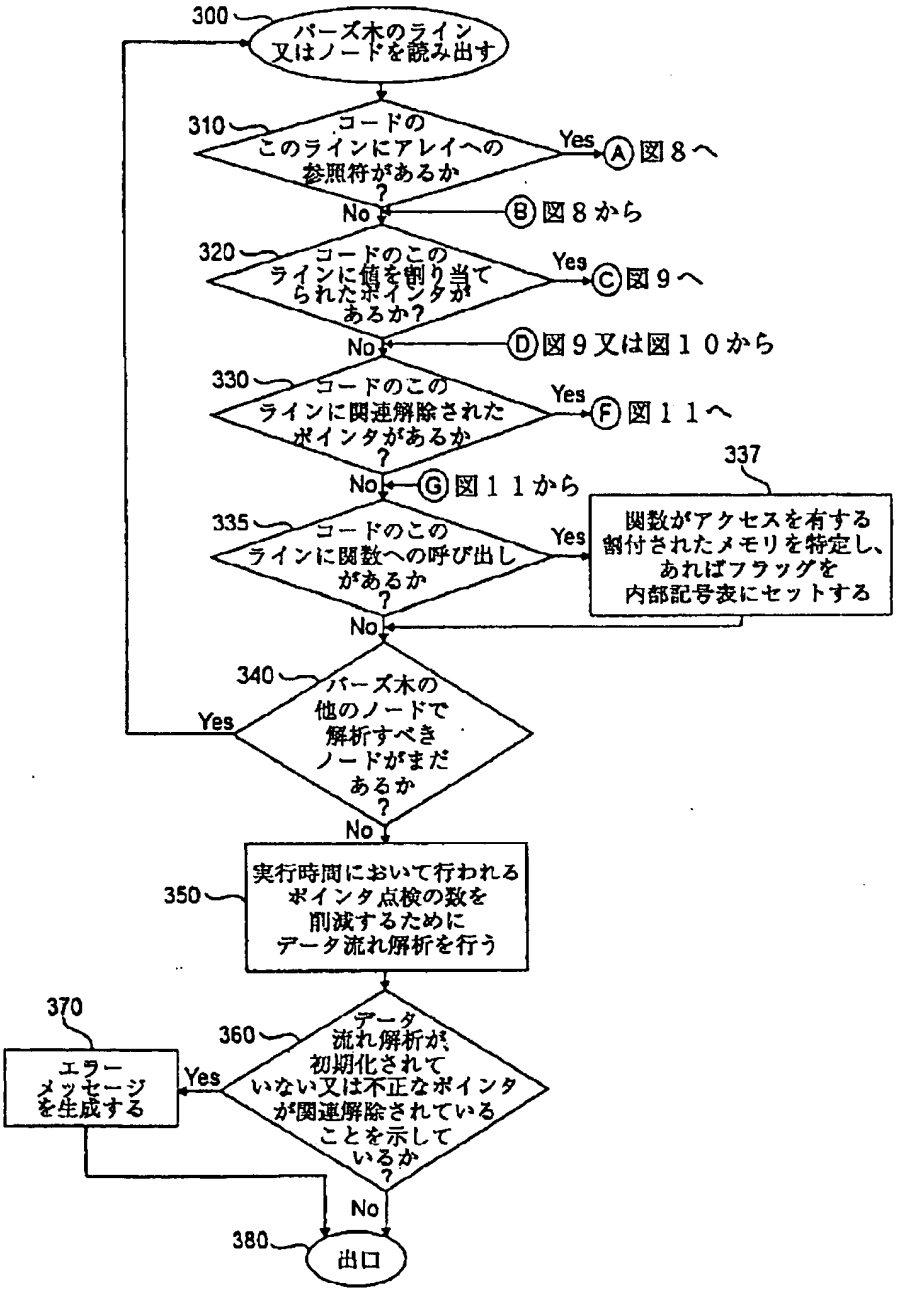
【図24】



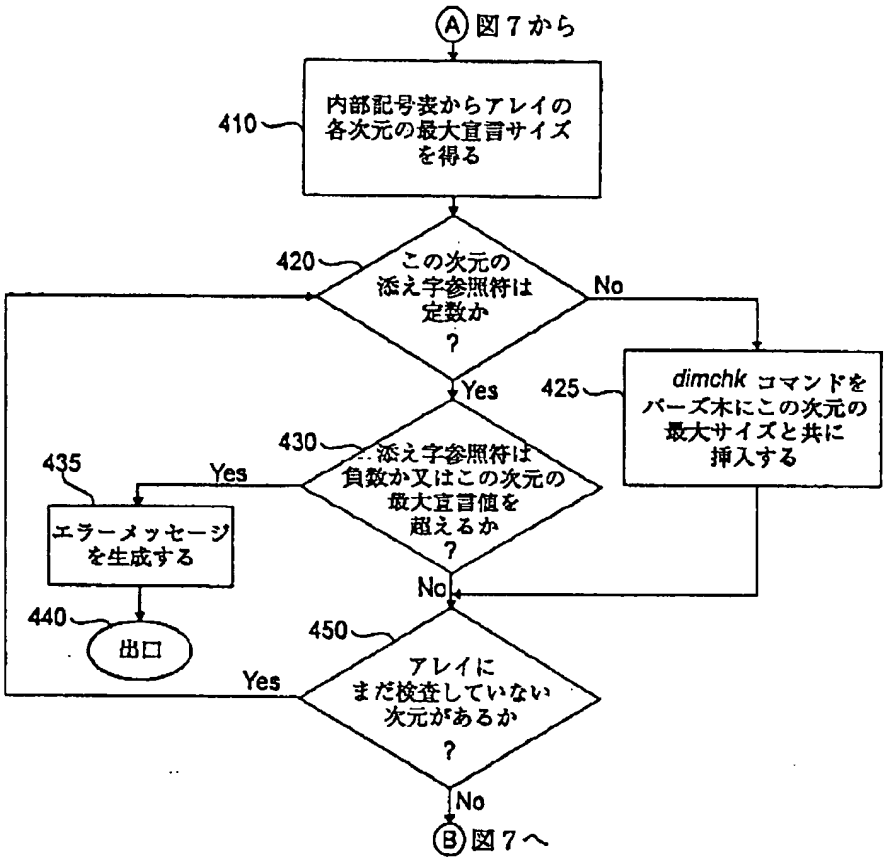
【図14】



【図７】


300
パーズ木のライン又はノードを読み出す
310
コードのこのラインにアレイへの参照符があるか？
Yes
Ⓐ 図８へ
No
Ⓑ 図８から
320
コードのこのラインに値を割り当てられたポインタがあるか？
Yes
Ⓒ 図９へ
No
Ⓓ 図９又は図１０から
330
コードのこのラインに関連解除されたポインタがあるか？
Yes
Ⓕ 図１１へ
No
Ⓖ 図１１から
337
関数がアクセスを有する割付されたメモリを特定し、あればフラッグを内部記号表にセットする
335
コードのこのラインに関数への呼び出しがあるか？
Yes
No
340
パーズ木の他のノードで解析すべきノードがまだあるか？
Yes
No
350
実行時間において行われるポインタ点検の数を削減するためにデータ流れ解析を行う
370
エラーメッセージを生成する
360
データ流れ解析が、初期化されていない又は不正なポインタが関連解除されていることを示しているか？
Yes
No
380
出口

【図８】

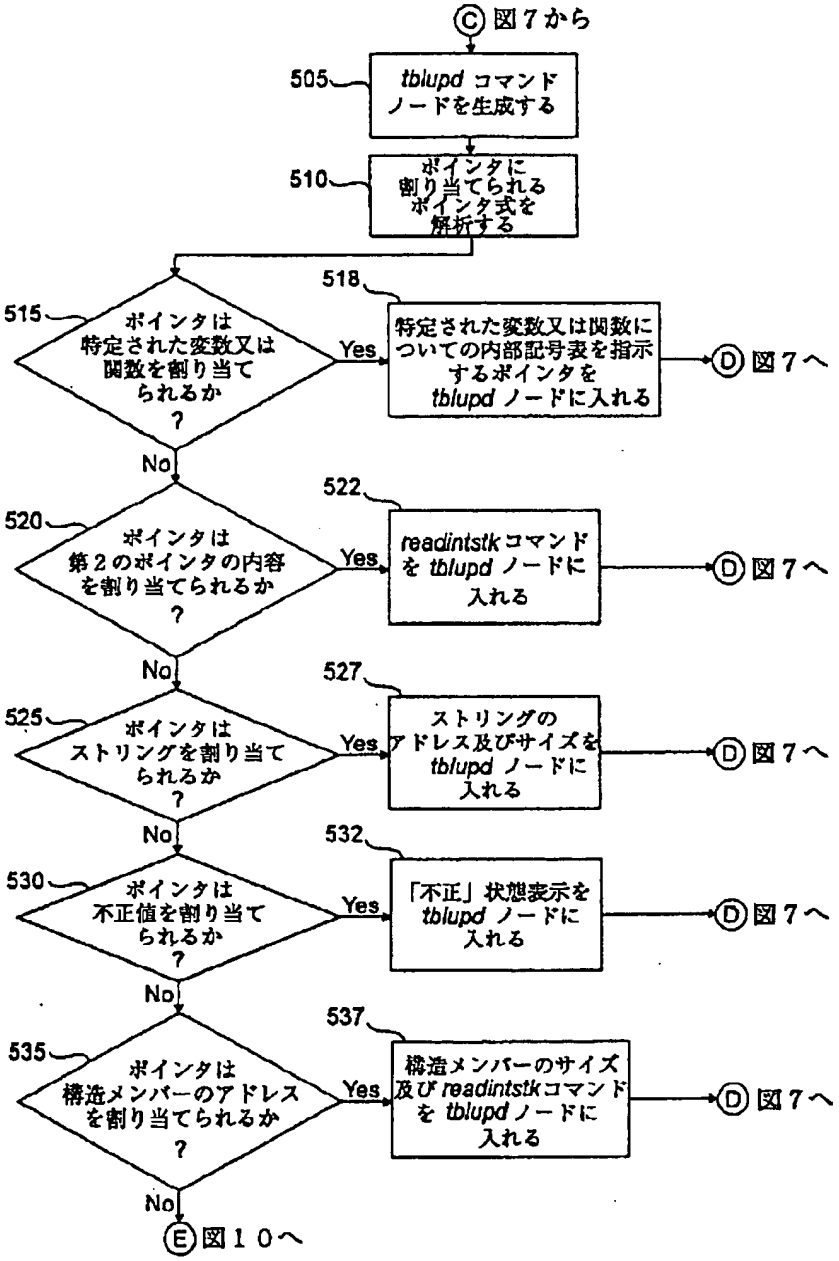


【図２３】

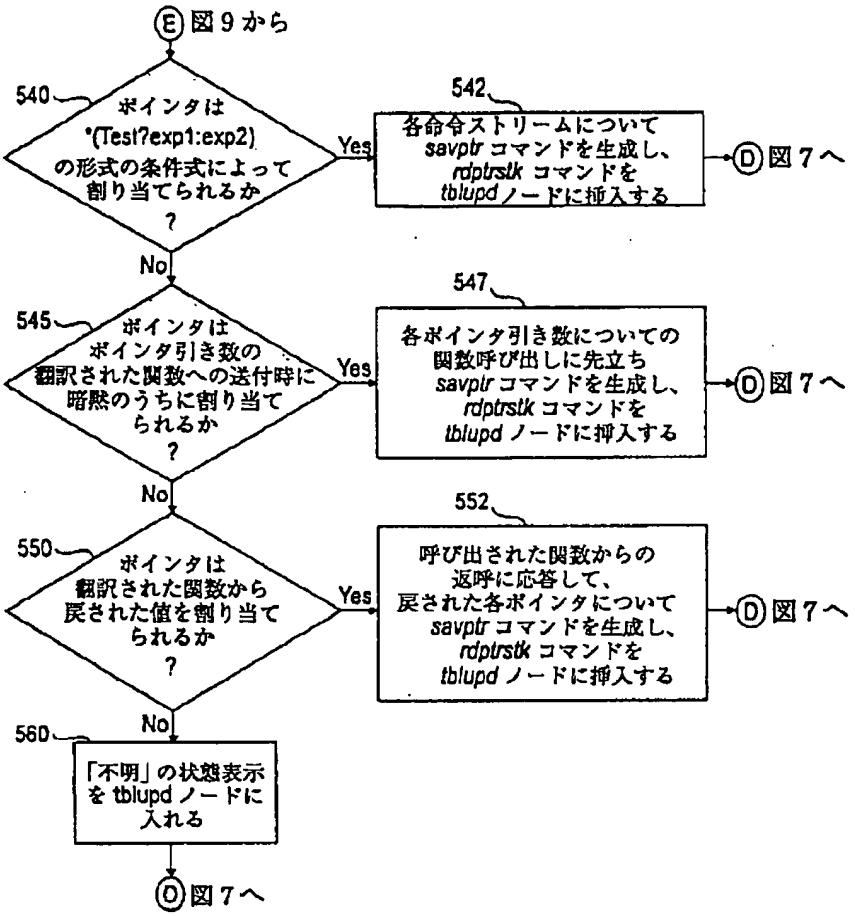
原始コードファイル「サンプル」

```
    •
    •
    •
 110    struct part {
 120    int number;
 130    char name [10];
 140    char color;
 150    } widget;
 160    struct part *ptr_part;
 170    ptr_part = &widget;
 180    ptr_part→number = 880;
 190    strcpy ( ptr_part→name "chair");
 1100   ptr_part→color ='b';
    •
    •
    •
 1150   ptr_part = 0;
 1160   ptr_part→number = 870;
    •
    •
    •
```

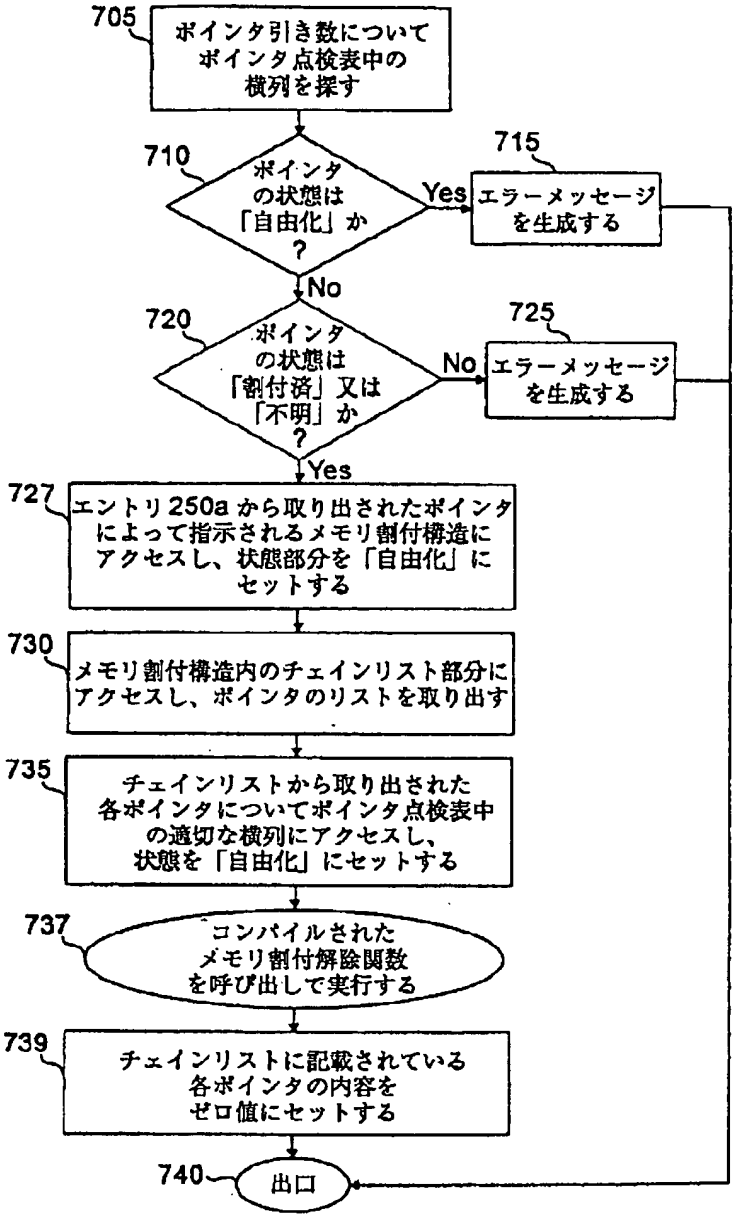
【図9】


Ⓒ 図7から
505
tblupd コマンドノードを生成する
510
ポインタに割り当てられるポインタ式を解析する
515
ポインタは特定された変数又は関数を割り当てられるか？
Yes
518
特定された変数又は関数についての内部記号表を指示するポインタをtblupd ノードに入れる
Ⓓ 図7へ
No
520
ポインタは第2のポインタの内容を割り当てられるか？
Yes
522
readintstkコマンドを tblupd ノードに入れる
Ⓓ 図7へ
No
525
ポインタはストリングを割り当てられるか？
Yes
527
ストリングのアドレス及びサイズをtblupd ノードに入れる
Ⓓ 図7へ
No
530
ポインタは不正値を割り当てられるか？
Yes
532
「不正」状態表示をtblupd ノードに入れる
Ⓓ 図7へ
No
535
ポインタは構造メンバーのアドレスを割り当てられるか？
Yes
537
構造メンバーのサイズ及び readintstkコマンドを tblupd ノードに入れる
Ⓓ 図7へ
No
Ⓔ 図10へ

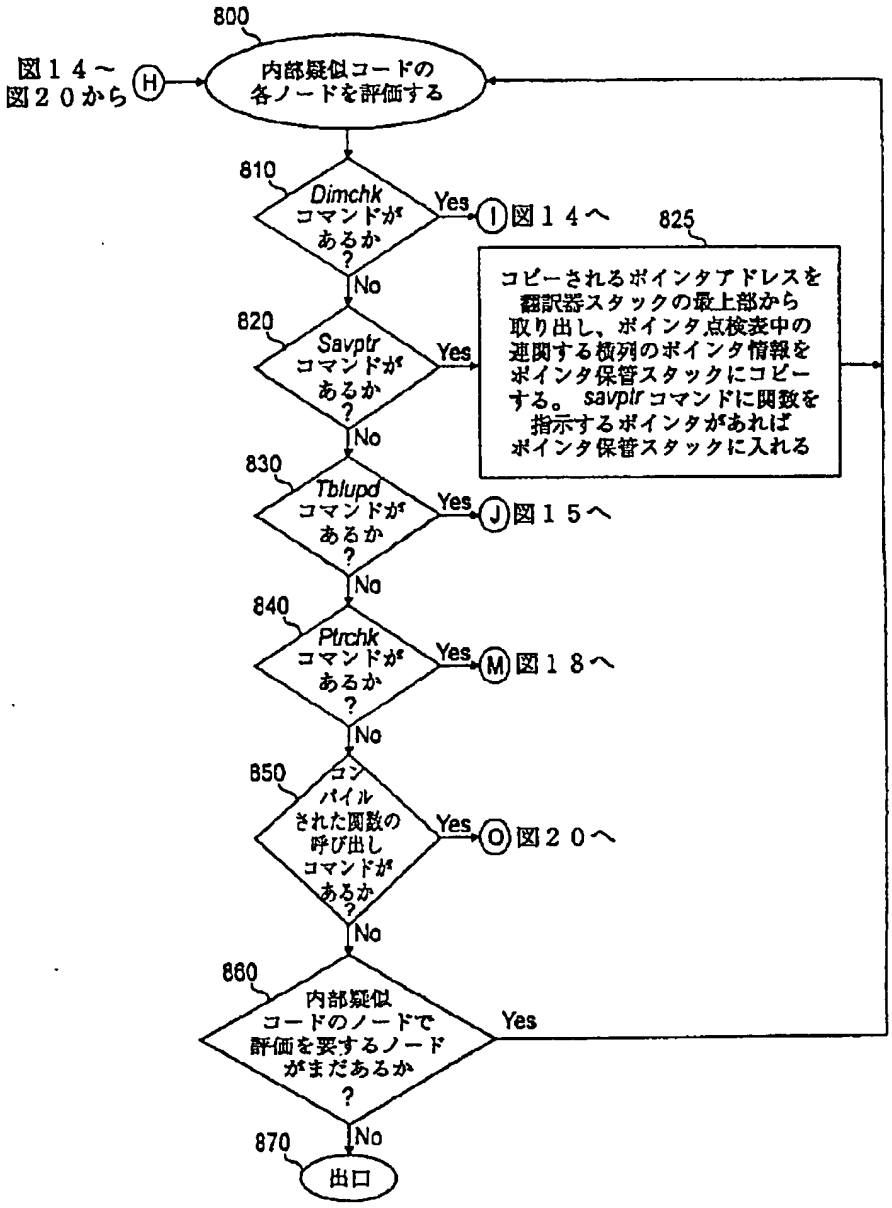
【図１０】


Ⓔ 図９から
540
ポインタは
*(Test?exp1:exp2)
の形式の条件式によって
割り当てられるか
?
Yes
542
各命令ストリームについて
savptr コマンドを生成し、
rdptrstk コマンドを
tblupd ノードに挿入する
Ⓓ 図７へ
No
545
ポインタは
ポインタ引き数の
翻訳された関数への送付時に
暗黙のうちに割り当て
られるか
?
Yes
547
各ポインタ引き数についての
関数呼び出しに先立ち
savptr コマンドを生成し、
rdptrstk コマンドを
tblupd ノードに挿入する
Ⓓ 図７へ
No
550
ポインタは
翻訳された関数から
戻された値を割り当て
られるか
?
Yes
552
呼び出された関数からの
返呼に応答して、
戻された各ポインタについて
savptr コマンドを生成し、
rdptrstk コマンドを
tblupd ノードに挿入する
Ⓓ 図７へ
No
560
「不明」の状態表示
を tblupd ノードに
入れる
Ⓓ 図７へ

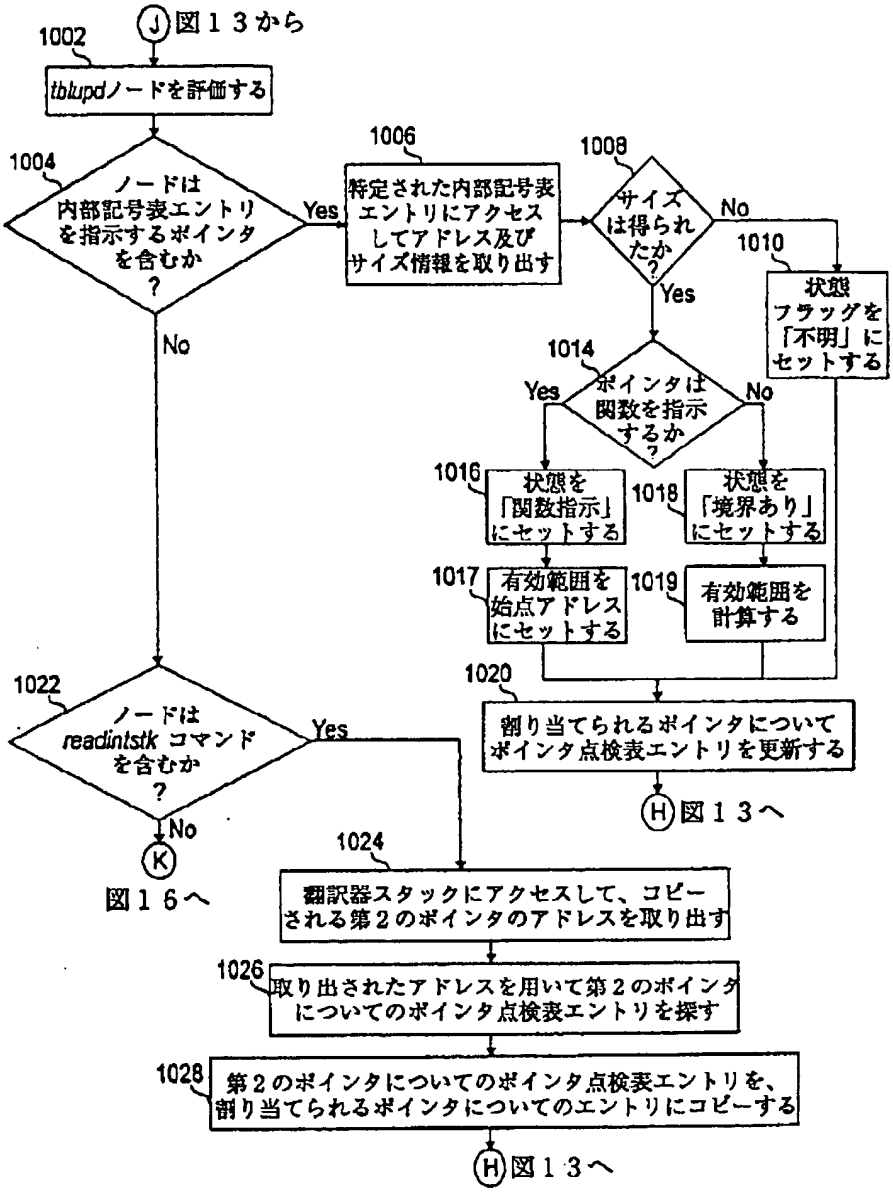
【図１２】


705
ポインタ引き数について
ポインタ点検表中の
横列を探す
710
ポインタ
の状態は
「自由化」か
？
Yes
715
エラーメッセージ
を生成する
No
720
ポインタ
の状態は
「割付済」又は
「不明」か
？
No
725
エラーメッセージ
を生成する
Yes
727
エントリ250aから取り出されたポインタ
によって指示されるメモリ割付構造に
アクセスし、状態部分を「自由化」に
セットする
730
メモリ割付構造内のチェインリスト部分に
アクセスし、ポインタのリストを取り出す
735
チェインリストから取り出された
各ポインタについてポインタ点検表中
の適切な横列にアクセスし、
状態を「自由化」にセットする
737
コンパイルされた
メモリ割付解除関数
を呼び出しで実行する
739
チェインリストに記載されている
各ポインタの内容を
ゼロ値にセットする
740
出口

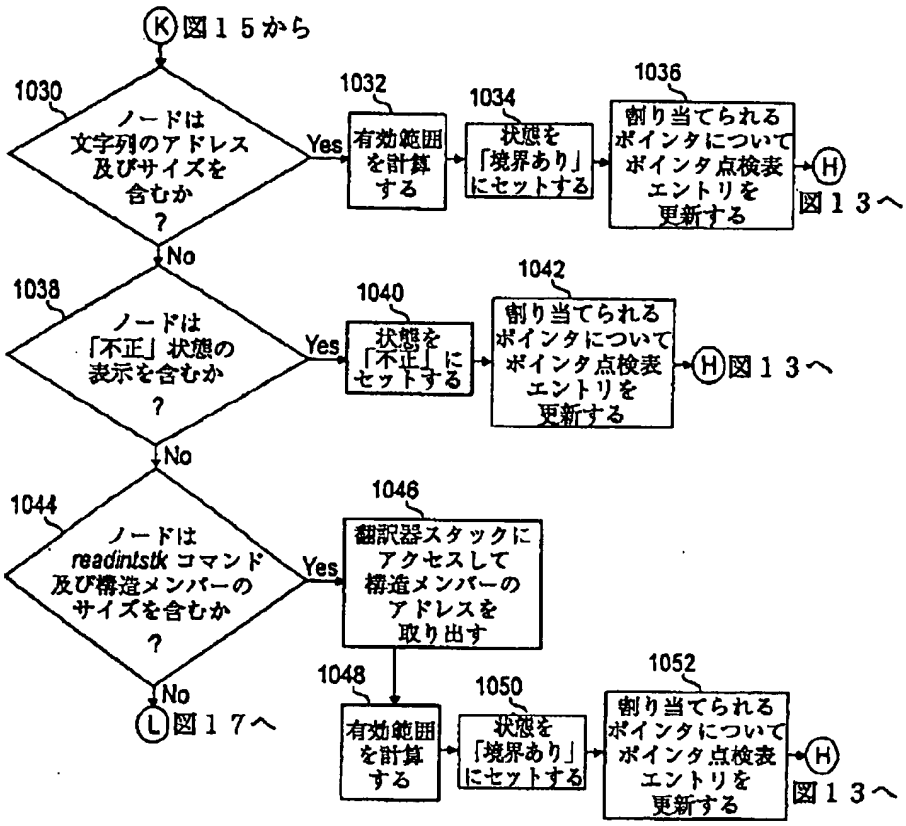
【図１３】


800
図１４～図２０から
H
内部疑似コードの各ノードを評価する
810
Dimchk コマンドがあるか？
Yes
I 図１４へ
No
825
コピーされるポインタアドレスを翻訳器スタックの最上部から取り出し、ポインタ点検表中の連関する横列のポインタ情報をポインタ保管スタックにコピーする。 savptr コマンドに関数を指示するポインタがあればポインタ保管スタックに入れる
820
Savptr コマンドがあるか？
Yes
No
830
Tblupd コマンドがあるか？
Yes
J 図１５へ
No
840
Ptrchk コマンドがあるか？
Yes
M 図１８へ
No
850
コンパイルされた関数の呼び出しコマンドがあるか？
Yes
O 図２０へ
No
860
内部疑似コードのノードで評価を要するノードがまだあるか？
Yes
No
870
出口

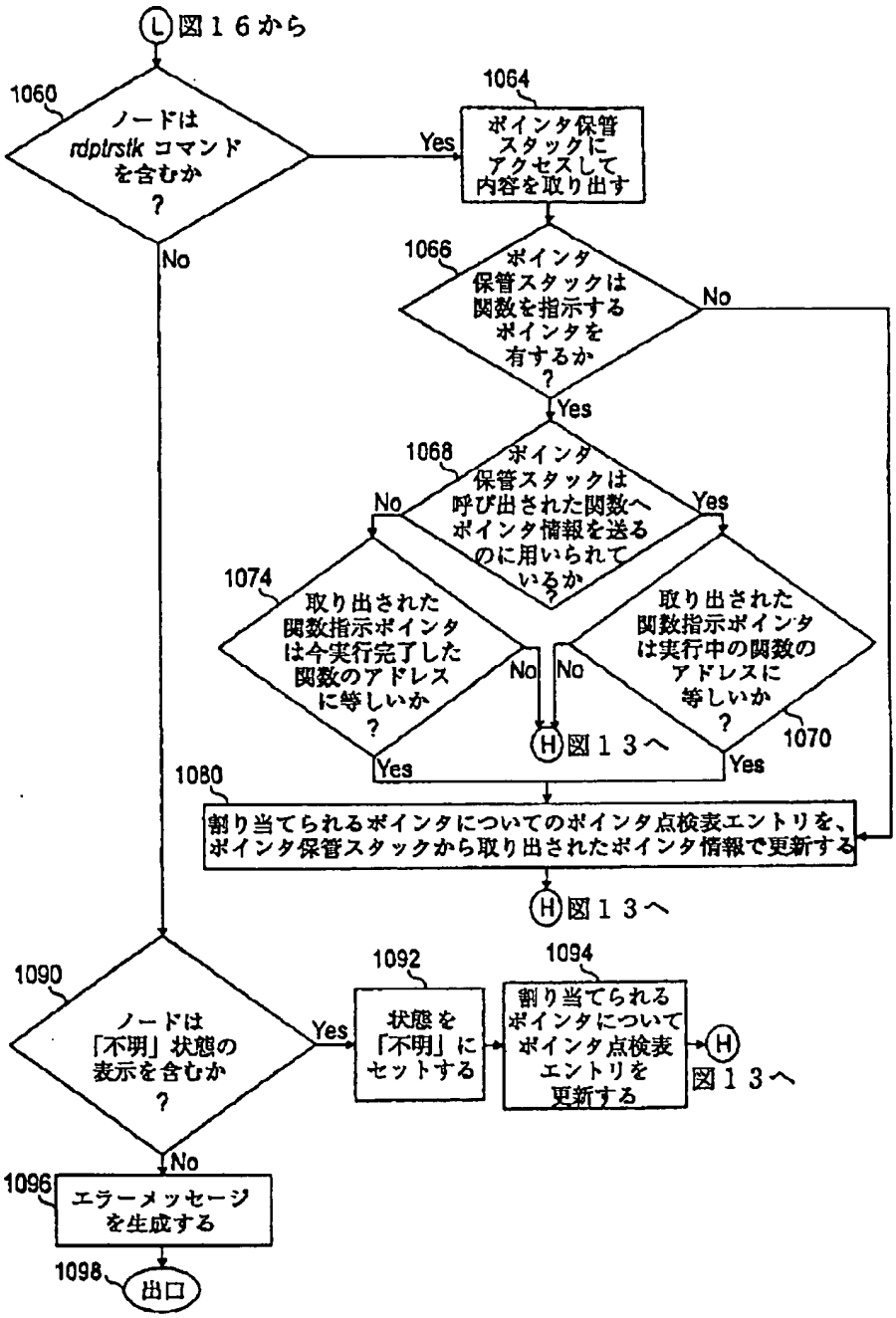
【図１５】


Ⓙ図１３から
1002
tblupdノードを評価する
1004
ノードは内部記号表エントリを指示するポインタを含むか？
Yes
No
1006
特定された内部記号表エントリにアクセスしてアドレス及びサイズ情報を取り出す
1008
サイズは得られたか？
No
Yes
1010
状態フラッグを「不明」にセットする
1014
ポインタは関数を指示するか？
Yes
No
1016
状態を「関数指示」にセットする
1018
状態を「境界あり」にセットする
1017
有効範囲を始点アドレスにセットする
1019
有効範囲を計算する
1020
割り当てられるポインタについてポインタ点検表エントリを更新する
Ⓗ図１３へ
1022
ノードはreadintstkコマンドを含むか？
Yes
No
Ⓚ
図１６へ
1024
翻訳器スタックにアクセスして、コピーされる第２のポインタのアドレスを取り出す
1026
取り出されたアドレスを用いて第２のポインタについてのポインタ点検表エントリを探す
1028
第２のポインタについてのポインタ点検表エントリを、割り当てられるポインタについてのエントリにコピーする
Ⓗ図１３へ

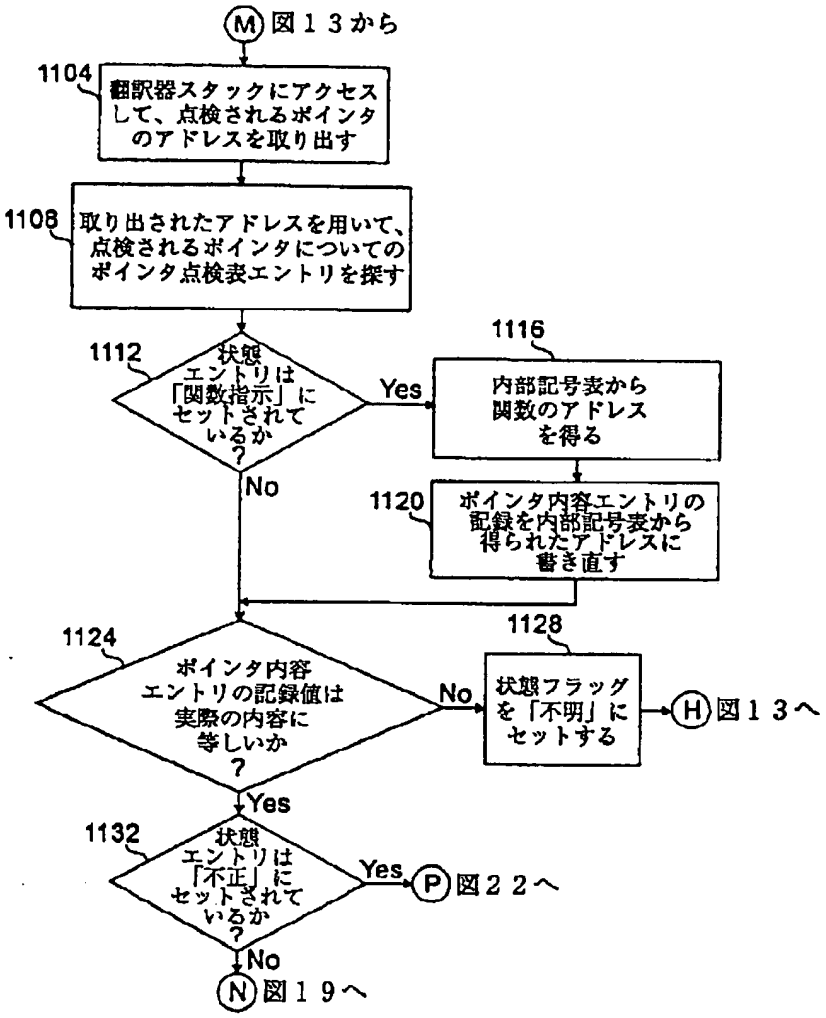
【図１６】


Ⓚ図１５から
1030
ノードは文字列のアドレス及びサイズを含むか？
Yes
1032
有効範囲を計算する
1034
状態を「境界あり」にセットする
1036
割り当てられるポインタについてポインタ点検表エントリを更新する
Ⓗ図１３へ
No
1038
ノードは「不正」状態の表示を含むか？
Yes
1040
状態を「不正」にセットする
1042
割り当てられるポインタについてポインタ点検表エントリを更新する
Ⓗ図１３へ
No
1044
ノードはreadintstkコマンド及び構造メンバーのサイズを含むか？
Yes
1046
翻訳器スタックにアクセスして構造メンバーのアドレスを取り出す
No
Ⓛ図１７へ
1048
有効範囲を計算する
1050
状態を「境界あり」にセットする
1052
割り当てられるポインタについてポインタ点検表エントリを更新する
Ⓗ図１３へ

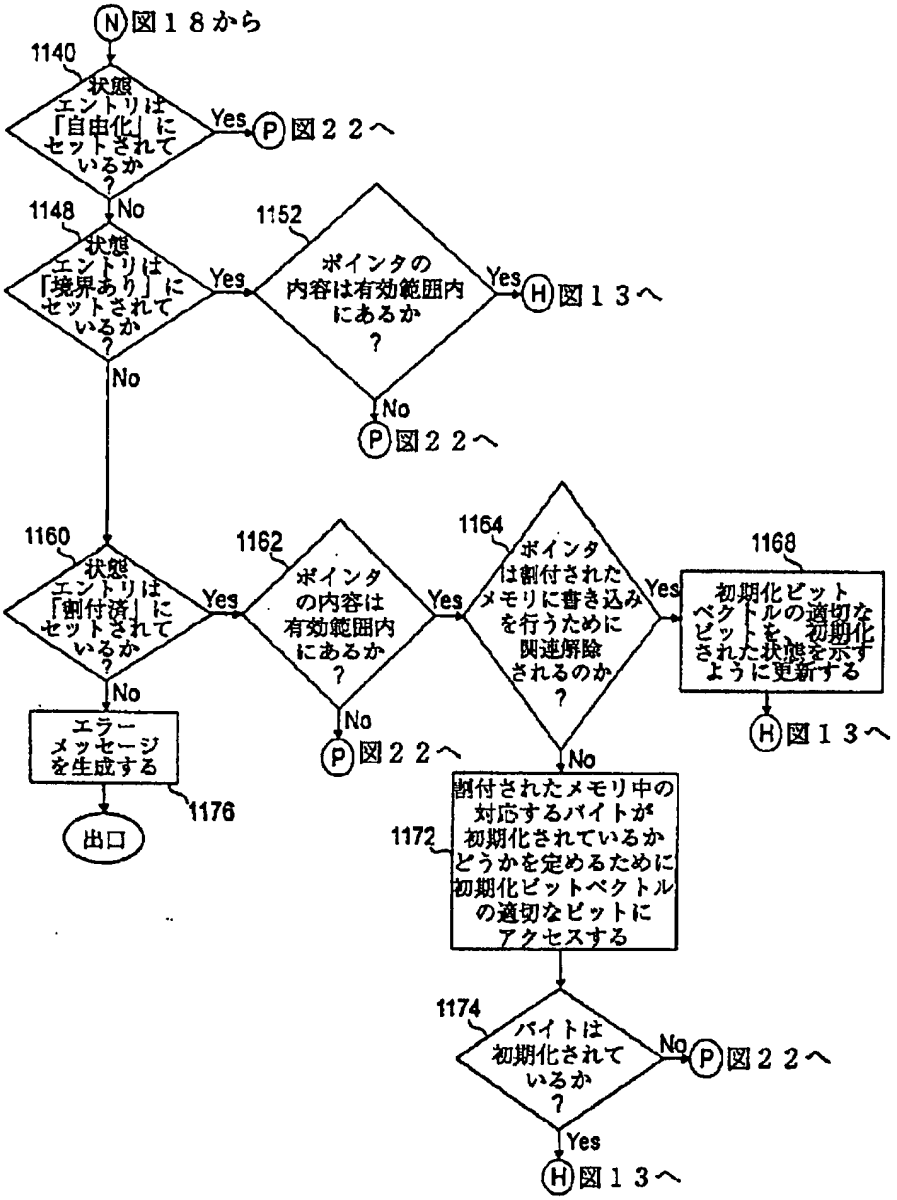
【図17】


Ⓛ図16から
1060
ノードは rdptrstk コマンドを含むか ?
Yes
No
1064
ポインタ保管スタックにアクセスして内容を取り出す
1066
ポインタ保管スタックは関数を指示するポインタを有するか ?
No
Yes
1068
ポインタ保管スタックは呼び出された関数へポインタ情報を送るのに用いられているか ?
No
Yes
1074
取り出された関数指示ポインタは今実行完了した関数のアドレスに等しいか ?
No
Yes
1070
取り出された関数指示ポインタは実行中の関数のアドレスに等しいか ?
No
Yes
Ⓗ図13へ
1080
割り当てられるポインタについてのポインタ点検表エントリを、ポインタ保管スタックから取り出されたポインタ情報で更新する
Ⓗ図13へ
1090
ノードは「不明」状態の表示を含むか ?
Yes
No
1092
状態を「不明」にセットする
1094
割り当てられるポインタについてポインタ点検表エントリを更新する
Ⓗ
図13へ
1096
エラーメッセージを生成する
1098
出口

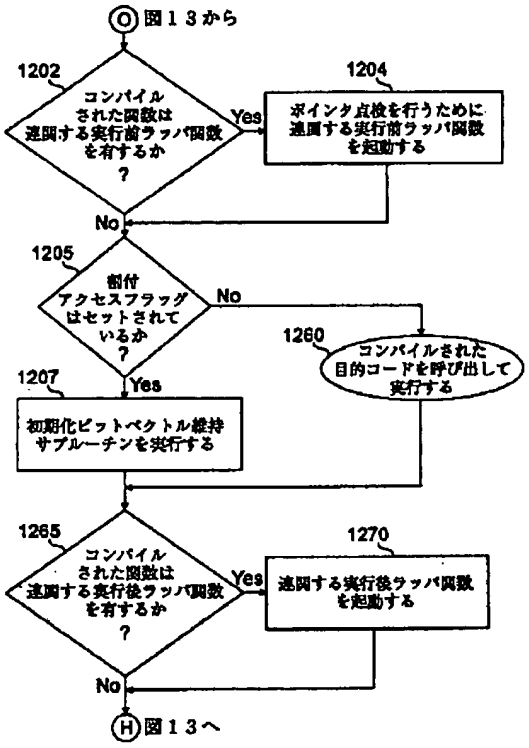
【図18】


M 図13から
1104
翻訳器スタックにアクセスして、点検されるポインタのアドレスを取り出す
1108
取り出されたアドレスを用いて、点検されるポインタについてのポインタ点検表エントリを探す
1112
状態エントリは「関数指示」にセットされているか？
Yes
1116
内部記号表から関数のアドレスを得る
No
1120
ポインタ内容エントリの記録を内部記号表から得られたアドレスに書き直す
1124
ポインタ内容エントリの記録値は実際の内容に等しいか？
No
1128
状態フラッグを「不明」にセットする
H 図13へ
Yes
1132
状態エントリは「不正」にセットされているか？
Yes
P 図22へ
No
N 図19へ

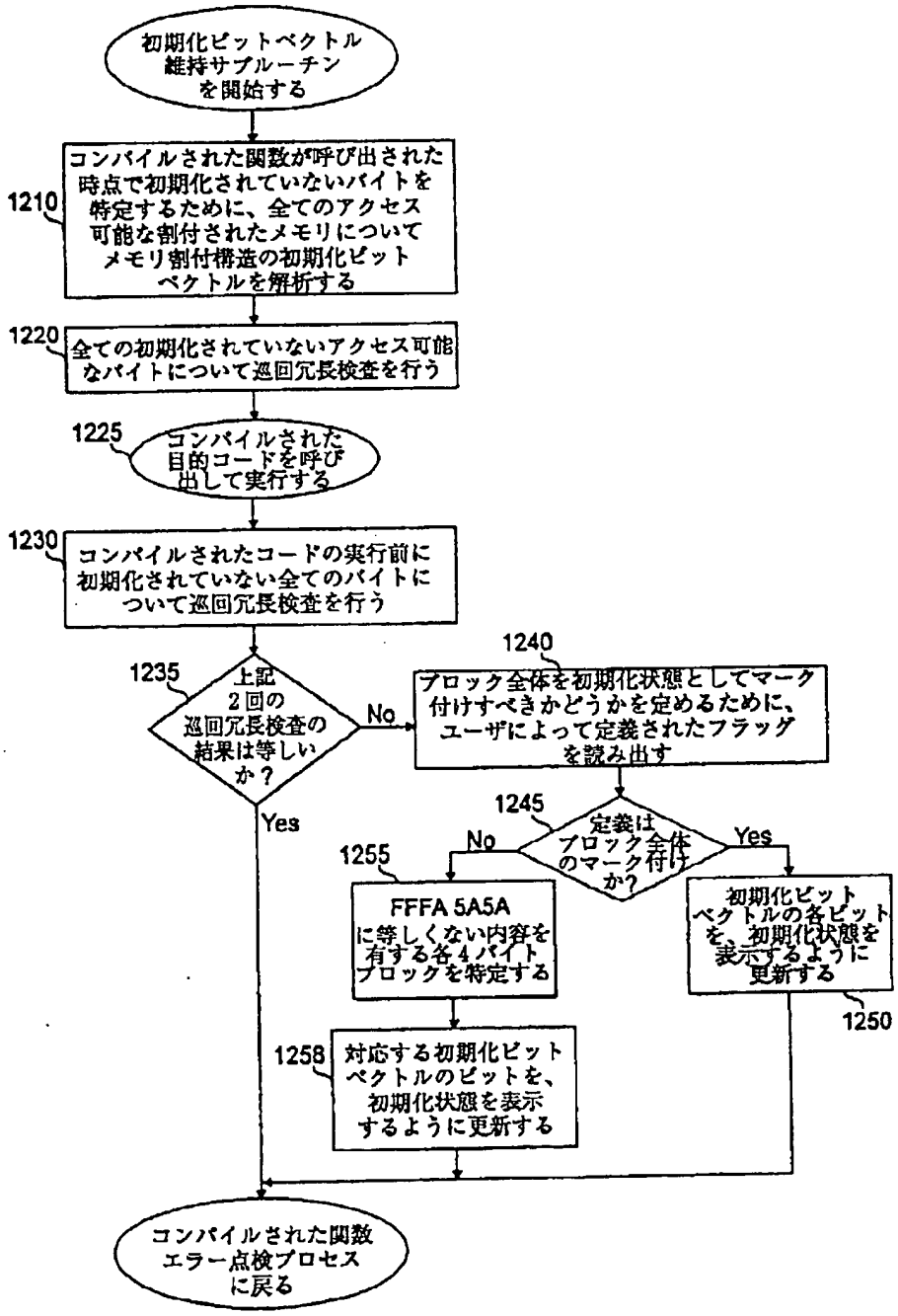
【図１９】


N 図１８から
1140
状態エントリは「自由化」にセットされているか？
Yes
P 図２２へ
No
1148
状態エントリは「境界あり」にセットされているか？
Yes
1152
ポインタの内容は有効範囲内にあるか？
Yes
H 図１３へ
No
P 図２２へ
No
1160
状態エントリは「割付済」にセットされているか？
Yes
1162
ポインタの内容は有効範囲内にあるか？
Yes
1164
ポインタは割付されたメモリに書き込みを行うために関連解除されるのか？
Yes
1168
初期化ビットベクトルの適切なビットを、初期化された状態を示すように更新する
H 図１３へ
No
P 図２２へ
No
エラーメッセージを生成する
1176
出口
No
1172
割付されたメモリ中の対応するバイトが初期化されているかどうかを定めるために初期化ビットベクトルの適切なビットにアクセスする
1174
バイトは初期化されているか？
No
P 図２２へ
Yes
H 図１３へ

【図２０】


Ⓞ 図１３から
1202
コンパイル された関数は 遅関する実行前ラッパ関数 を有するか ？
Yes
1204
ポインタ点検を行うために 遅関する実行前ラッパ関数 を起動する
No
1205
割付 アクセスフラグ はセットされて いるか ？
No
1260
コンパイルされた 目的コードを呼び出して 実行する
1207
Yes
初期化ビットベクトル維持 サブルーチンを実行する
1265
コンパイル された関数は 遅関する実行後ラッパ関数 を有するか ？
Yes
1270
遅関する実行後ラッパ関数 を起動する
No
Ⓗ 図１３へ

【図２１】



フロントページの続き

(72)発明者　サディアス　ジュリアス　コワルスキー
アメリカ合衆国、07901　ニュージャージー、サミット、ストーンリッジ　ロード　73

(72)発明者　ジェームズ　アール．ローランド
アメリカ合衆国、07078　ニュージャージー、ショート　ヒルズ、サッカレー　ドライブ　18